Abbildung sequentieller C-Programme auf parallel arbeitende Zellularautomaten

Dissertation

zur Erlangung des Doktorgrades
der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt der

Fakultät für Mathematik/Informatik und Machinenbau
der Technischen Universität Clausthal

am 23.06.2015

von

Jens Drieseberg

geboren am 23.01.1984 in Erfurt

Gutachter:
Prof. Dr. rer. nat. Christian Siemers
Prof. Dr. rer. nat. Sven Hartmann

Datum der mündlichen Prüfung: 24.08.2016

Zusammenfassung

Single-Core Prozessorsysteme werden in der Gegenwart zunehmend von Multi-Core Architekturen abgelöst. Während bis zu Beginn der 2000er Jahre parallele Architekturen hauptsächlich in wissenschaftlichen und industriellen Großrechnern eingesetzt wurden, findet man heute in vielen verschiedenen Gerätetypen Multi-Core Prozessoren - angefangen bei Heimcomputern bzw. Arbeitsplatzrechnern, über Smartphones und Tablets bis hin zu eingebetteten Systemen in Fahrzeugen.

Ein Problem paralleler Architekturen besteht aus der Sicht von Softwareentwicklern in der, verglichen mit Single-Core Systemen, aufwändigen Programmierung. Zur effizienten Ausnutzung aller zur Verfügung stehenden Ressourcen müssen Konzepte erarbeitet werden, die einerseits möglichst viele Berechnungen parallel durchführen, andererseits keine Fehler aufgrund von Dateninkonsistenzen zwischen den Recheneinheiten erzeugen. Hinzu kommt, dass zur Programmierung verschiedener paralleler Architekturen unterschiedliche Konzepte angewendet und voneinander abweichende Programmiermodelle eingesetzt werden müssen. Ein höherer Entwicklungsaufwand und zusätzlichen Kosten bei der Erstellung von Software sind Folgen davon.

Die vorliegende Arbeit widmet sich der Lösung dieser Probleme und bietet Lösungsansätze für zukünftige Untersuchungen. Es wird ein Verfahren zur Abbildung sequentieller Algorithmen, geschrieben in der Programmiersprache C, auf ein Modell vorgestellt, das auf Zellularautomaten basiert. Die Zellen des Modells und ihre Verbindungen untereinander stellen eine Repräsentation des kompilierten Algorithmus dar und sind in der Lage, diesen durch Bearbeitung von Teilaufgaben und Kommunikation untereinander auszuwerten. Das entwickelte System ist selbstsynchronisierend, das bedeutet, die Zellen können parallel verarbeitet werden und beachten selbständig die durch den Algorithmus implizit vorgegebenen Datenabhängigkeiten. Im Gegensatz zu anderen bereits existierenden Verfahren ist die Übersetzung der Algorithmen nicht auf eine bestimmte Zielarchitektur ausgerichtet. Die aus den Algorithmen generierten Zellularautomaten können auf verschiedene parallele Architekturen übertragen und auf diesen Systemen ausgewertet werden.

Zur Bearbeitung der generierten Zellularautomaten sind sowohl Multi-Core CPU-Systeme, Many-Core GPU-Systeme als auch FPGAs einsetzbar. In der Arbeit werden Verfahren zur Übertragung und Auswertung auf die genannten Architekturen beschrieben sowie experimentelle Vergleiche zur Performance der Zellularautomaten auf den verschiedenen Systemen vorgestellt.

Danksagung

Mein erster Dank geht an meinen Betreuer Prof. Dr. Christian Siemers, der mir geholfen hat, ein passendes Thema für meine Arbeit zu finden und mich während der gesamten Zeit beraten und unterstützt hat.

Mein weiterer Dank gilt Prof. Dr. Sven Hartmannn für seine Tätigkeit als Zweitgutachter sowie für seine Anmerkungen und Korrekturvorschläge.

Prof. Dr. Jörg P. Müller danke ich für seine Unterstützung bei der Finanzierung meiner Stelle am Institut für Informatik.

Des Weiteren bedanke ich mich bei allen Mitarbeiterinnen und Mitarbeitern des Instituts für Informatik für die interessante und gute Zusammenarbeit. Insbesondere gilt mein Dank Danilo Gasdzik, Dr. Andreas Harrer, Dr. Michaela Huhn, Michael Köster, Jürgen Lorenz, Dr. David Mainzer, Dr. Andreas Reinhardt, Dr. René Weller und Sabrina Wittek.

René Fritzsche und Sascha Lützel danke ich für die Aufnahme in die Embedded Systems Gruppe und das dazugehörige Büro, für die gute Zusammenarbeit bei verschiedensten Themen und Projekten sowie ihre hilfreichen Ideen.

Prof. Dr. Kai Hormann und Dr. Tim Winkler danke ich für den Vorschlag, im Anschluss an mein Studium eine Promotion anzugehen. Ohne sie hätte ich das sicher nie umgesetzt. Insbesondere bei Tim bedanke ich mich für die Erklärungen und Tipps zur Programmierung sowie zur wissenschaftlichen Herangehensweise an Themen, die mir auf meinem Weg sehr geholfen haben.

Bei meinen Eltern bedanke ich mich dafür, dass sie mir meine Ausbildung ermöglicht haben und für die jederzeit zur Verfügung stehende Hilfe.

Nicht zuletzt danke ich meiner Frau Lucienne und meinen Töchtern Zoe und Elena für ihr Verständnis, die Unterstützung und die Motivation die sie mir während der gesamten Zeit gegeben haben.

Inhaltsverzeichnis

Abkürzungen

ANSI	-	American National Standards Institute
ASIC	-	Application-Specific Integrated Circuit
CPU	-	Central Processing Unit
FPGA	-	Field Programmable Gate Array
FPU	-	Floating-Point Unit
GPU	-	Graphics Processing Unit
IP	-	Intellectual Property
LU	-	Loop Unrolling
LUT	-	Lookup Table
MIMD	-	Multiple Instruction, Multiple Data
PCI	-	Peripheral Component Interconnect
SIMD	-	Single Instruction, Multiple Data
SISD	-	Single Instruction, Single Data
SSA	-	Static Single Assignment
VHDL	-	VHSIC Hardware Description Language
VHSIC	-	Very High Speed Integrated Circuit

1 Einleitung

Heutige Computer und eingebettete Systeme unterscheiden sich stark von den Systemen, die bis 2005 hauptsächlich eingesetzt wurden [86]. Bis zu diesem Zeitpunkt dominierten Einkernprozessoren den Massenmarkt. Höhere Rechenleistung wurde hauptsächlich durch eine höhere Taktfrequenz des Prozessors erreicht. Zur Erreichung eines höheren Taktes ist, unter sonst gleichen Bedingungen, mehr Energie notwendig und es wird mehr Abwärme erzeugt. Diese muss mit entsprechenden Gegenmaßnahmen abgeführt werden. Mit zunehmender Leistung ist das jedoch nur noch mit erheblichem technischem Aufwand zu bewerkstelligen, zum Beispiel durch Wasser- oder Stickstoffkühlung. Solche Systeme sind teuer und für den Massenmarkt ungeeignet. Die *Intel Coporation* hat aus den genannten Gründen, im Jahr 2004, die geplante Einführung des *Tejas Pentium 4* Prozessors aufgeben müssen [114]. Es mussten andere Methoden gefunden werden, um die Leistungsfähigkeit neuer Prozessoren zu steigern.

Eine seit 2006 genutzte Möglichkeit war die Einführung von *Dual-Core Prozessoren* [86]. Bei Dual-Core Prozessoren werden zwei vollwertige Rechenkerne auf einem Chip nebeneinander integriert. Selbst einfache PCs, Tablets und Smartphones haben mittlerweile meist mehr als einen Rechenkern in ihrer zentralen Recheneinheit (CPU) zur Verfügung. Aktuelle PC-Prozessoren *(Stand Oktober 2014)* haben bis zu acht Rechenkerne und können durch Prozessorvirtualisierung bis zu 16 Threads gleichzeitig verarbeiten (Intel Core i7-5960X Extreme Edition [57]).

Zu etwa der gleichen Zeit als Mehrkernprozessoren auf dem PC Markt auftauchten, entstanden auch Konzepte zur Nutzung von Grafikkarten für allgemeine Aufgaben. Die Konzepte, bezeichnet als *General Purpose Computation on Graphics Processing Unit (GPGPU)*, sind eine Erweiterung programmierbarer Shader. Obwohl ursprünglich für die parallele Verarbeitung von Pixeln konzipiert, erkannte man kurz nach der Jahrtausendwende, dass durch eine Erweiterung der Funktionalität diese speziellen Rechenwerke auch für die paralle-

le Verarbeitung anderer Aufgaben geeignet sind [111]. Die beiden großen Hersteller von Grafikkarten *Advanced Micro Devices, Inc. (AMD)* und *NVIDIA Corporation* bieten seit 2007 reine Rechenmodule mit aktuell bis zu 2880 parallelen Rechenwerken an (*NVIDIA Tesla K40* [81]).

Parallele Architekturen bieten, verglichen mit Single-Core Prozessoren, deutlich mehr Möglichkeiten in Bezug auf die mit ihnen erreichbare Performance. Andererseits stellen sie besondere Herausforderungen an die Softwareentwickler. Sie müssen ihre Denkweise anpassen und vom sequentiellen Programmfluss abweichen, um Programme zu entwickeln, die die Möglichkeiten paralleler Prozessoren ausnutzen. Insbesondere bei bereits existierenden Algorithmen ist das oftmals mit erheblichem Aufwand verbunden.

Vergleicht man die verschiedenen Architekturen und Programmiermodelle miteinander, so ergeben sich die folgenden grundlegenden Konzepte aus Sicht eines Programmierers:

System mit Single-Core CPU

Die Aufgaben des Programmierers betreffen hierbei ausschließlich die Planung des Programmflusses, da nur die Programmfunktionalität umgesetzt/codiert werden muss. Das Programm läuft in einem Thread und beansprucht die gesamte Leistung der CPU.

System mit Multi-Core CPU

Die Aufgaben des Programmierer betreffen, wie auch beim Single-Core System, die Planung des Programmflusses. Allerdings bedarf es bei Mehrkernsystemen einer komplexeren Planung. Einerseits muss die Programmfunktionalität gewährleistet sein, andererseits sollte der Programmfluss so geplant werden, dass die Arbeitslast möglichst auf alle Prozessoren abgebildet werden kann, um die volle Leistungsfähigkeit des Systems zu nutzen. Eine Gleichverteilung der Arbeitslast ist nur in seltenen Fällen möglich, da viele Programmteile sowohl auf Daten- als auch auf Algorithmenebene voneinander abhängig sind.

Durch diese Probleme ist es für den Programmierer notwendig, einen Hauptthread zu planen, der alle anderen Threads koordiniert. Der Hauptthread vergibt vom Programmierer vorgegebene Teilaufgaben an die anderen Threads und damit an die zur Verfügung stehenden Prozessoren. Das führt zwangsläu-

fig dazu, dass zusätzliche Synchronisationspunkte geplant werden müssen, um den Speicher bzw. die Daten zu synchronisieren und vor allem Datenkonsistenz zu garantieren.

Um Lese- bzw. Schreibfehler in den Daten zu vermeiden ist es notwendig, so genannte *Lock-Mechanismen* zu nutzen. Kritische Daten bzw. Variablen werden durch *Locks* beim Lesen bzw. Schreiben für andere Threads gesperrt, so dass diese warten müssen, bis die Daten wieder freigegeben werden. Mit diesem Verfahren kann der Programmierer die Datenkonsistenz zwischen den Threads garantierten. Ein unerwünschter Nebeneffekt von Locks ist die Gefahr von möglichen *Deadlocks*. Dabei blockieren sich Threads gegenseitig, indem sie auf die Freigabe einer Variablen des jeweils anderen Threads warten. Da dieser andere Thread die Variable nie freigibt, weil ihm ebenfalls Daten zur Bearbeitung fehlen, warten sie unendlich aufeinander. Ein weiterer negativer Effekt von Locks ist, dass das Warten zu einem Stillstand des Threads und damit zu einem Leistungsverlust des Systems führt.

System mit Single-Core/Multi-Core CPU und Many-Core GPU

Die Nutzung von Grafikkarten zur Ausführung allgemeiner Aufgaben bietet für den Programmierer den Vorteil einer großen Anzahl von gleichzeitig ausführbaren Threads und der damit verbunden Leistungsfähigkeit des Systems. Die einzelnen Kerne einer Grafikkarte sind zwar langsamer getaktet und damit einzeln nicht so leistungsfähig wie CPU Rechenkerne, allerdings sind sie in viel größerer Stückzahl vorhanden. Dadurch wird das Leistungsdefizit bei entsprechender Programmierung problemlos ausgeglichen [61].

Die Schwierigkeit bei Berechnungen auf der GPU besteht, vergleichbar mit den Problemen bei Multi-Core CPU Systemen, in der Verteilung der Aufgaben an die einzelnen Threads. Die Aufteilung auf die einzelnen Kerne der CPU und GPU muss vom Programmierer genau geplant werden, um ein möglichst performantes Programm zu erzeugen. Des Weiteren haben die GPU und CPU keinen gemeinsamen Speicher, so dass Daten, die für die Berechnung notwendig sind, zwischen den Arbeitsspeichern ausgetauscht werden müssen. Durch den Datenaustausch kann sich ein Problem für die Performance ergeben, wenn dies, im Vergleich zur Berechnungszeit, sehr häufig geschehen muss.

System mit programmierbarer Hardware

Neben den grundlegenden Konzepten der performanceorientieren Softwareentwicklung für Standard-Computer mit CPU und Grafikkarte, werden auch spezielle Hardwarelösungen für die algorithmische Verarbeitung von Daten eingesetzt. Einen möglichen Ansatz dazu bietet die Programmierung von *Field Programmable Gate Arrays (FPGA)*.

Programmierer entwickeln hierbei mithilfe einer speziellen Hardwarebeschreibungssprache eine logische Schaltung, die anschließend auf einen integrierten Schaltkreis programmiert wird. Anders als bei der Softwareentwicklung üblich, arbeiten diese logischen Schaltungen voll parallel. Entwickler müssen explizit Reihenfolgen vorgeben, um an von ihnen gewünschten (für den Algorithmus notwendigen) Stellen sequentielle Abarbeitung zu erzwingen. Eine Besonderheit ist zudem, dass Entwickler bei ihrem Design neben der parallelen Abarbeitung auch den Takt und die Größe ihres „Programms“ zu beachten haben. Da ein FPGA, verglichen mit einem Computer, über sehr begrenzte Ressourcen verfügt [93], ist hierauf besonderes Augenmerk zu legen.

Entsprechend programmierte FPGAs können beispielsweise mit einem PC über PCI-Express oder Netzwerk gekoppelt werden und für performancekritische Berechnungen als Beschleuniger eingesetzt werden. Das Verfahren ähnelt dabei dem der Nutzung von GPUs als Beschleuniger. Die zu verarbeitenden Daten müssen zwischen der CPU und dem FPGA ausgetauscht werden, da kein gemeinsamer Speicher existiert.

Der Vorteil von FPGAs besteht darin, dass es sich um Hardware handelt, bei der alle Wege durch die Programmierung festgelegt sind. Das macht sie, im Bezug auf ihre Performance, den Softwarelösungen überlegen. Des Weiteren sind sie energieeffizienter als CPU oder GPU Prozessoren, da sie mit einem deutlich geringeren Takt laufen und somit weniger Strom und keine aufwändigen Kühltechniken benötigen [63]. Ihr großer Nachteil ist, die sich stark von einer Softwarelösung unterscheidende Programmierung, was es für Softwareentwickler schwierig macht, sie einzusetzen. Neben der reinen Abbildung der Funktionalität, wie das bei Softwareentwicklung der Fall ist, muss zuerst die auf den Anwendungsfall spezialisierte Systemstruktur entworfen werden. In einem anschließenden Schritt kann dann die Abbildung des Anwendungsprozesses auf diese Systemstruktur erfolgen [64].

Zielsetzung der Arbeit

Mit dieser Arbeit soll nachgewiesen werden, dass eine auf *Zellularautomaten* basierende Repräsentation zur automatischen Übersetzung sequentieller Programme für parallele Architekturen geeignet ist. Dadurch wird es möglich, die vorgestellten Konzepte zu vereinheitlichen und den Planungsaufwand für Softwareentwickler auf das grundlegende Konzept einer Anwendung für Single-Core-CPU Systeme zu reduzieren. Der generierte Zellularautomat berechnet den in ANSI C-Code programmierten Algorithmus und ist auf allen beschriebenen Zielsystemen lauffähig, ohne dass der Entwickler Kenntnisse über parallele Programmierung oder über die zugrundeliegende Hardware haben muss. Die universelle Ausführbarkeit auf verschiedenen Zielarchitekturen unterscheidet das entwickelte System von bereits existierenden Verfahren zur Parallelisierung von C-Programmen, die auf nur eine bestimmte Zielarchitektur ausgerichtet sind.

Für bereits existierende Algorithmen eröffnet sich durch die Abbildung der Algorithmen auf Zellularautomaten die Möglichkeit, die Leistungsfähigkeit moderner paralleler Systeme zu nutzen. Die kompilierten Programme profitieren zudem von der Skalierbarkeit der Laufzeitumgebung. Schnellere, vor allem aber eine höhere Anzahl an zur Verfügung stehenden Prozessoren führt bei ausreichend großen Programmen ohne erneutes Kompilieren zu einer verbesserten Performance der übersetzten Algorithmen.

Abbildung 1.1 gibt einen Überblick über das in der vorliegenden Arbeit beschriebene System.

Der Aufbau der vorliegenden Arbeit gliedert sich wie folgt: Kapitel 2 gibt einen Überblick zu bereits existierenden Verfahren zur Parallelisierung sequentieller Programme und stellt Zellularautomaten im klassischen Sinn sowie zur Bearbeitung allgemeiner Probleme vor.

Kapitel 3 beschreibt die Implementierung des Compilers - vom Einlesen des Quellcodes bis zur Abbildung der Algorithmen auf Zellen. Des Weiteren wird die Umsetzung der Laufzeitumgebung zur Auswertung der Zellularautomaten auf der CPU, der GPU und dem FPGA erläutert.

Erweiterungen des Basis-Ansatzes um zusätzliche Funktionalitäten und die Abbildung komplexerer, in C-Algorithmen eingesetzter, Konzepte auf Zellula-

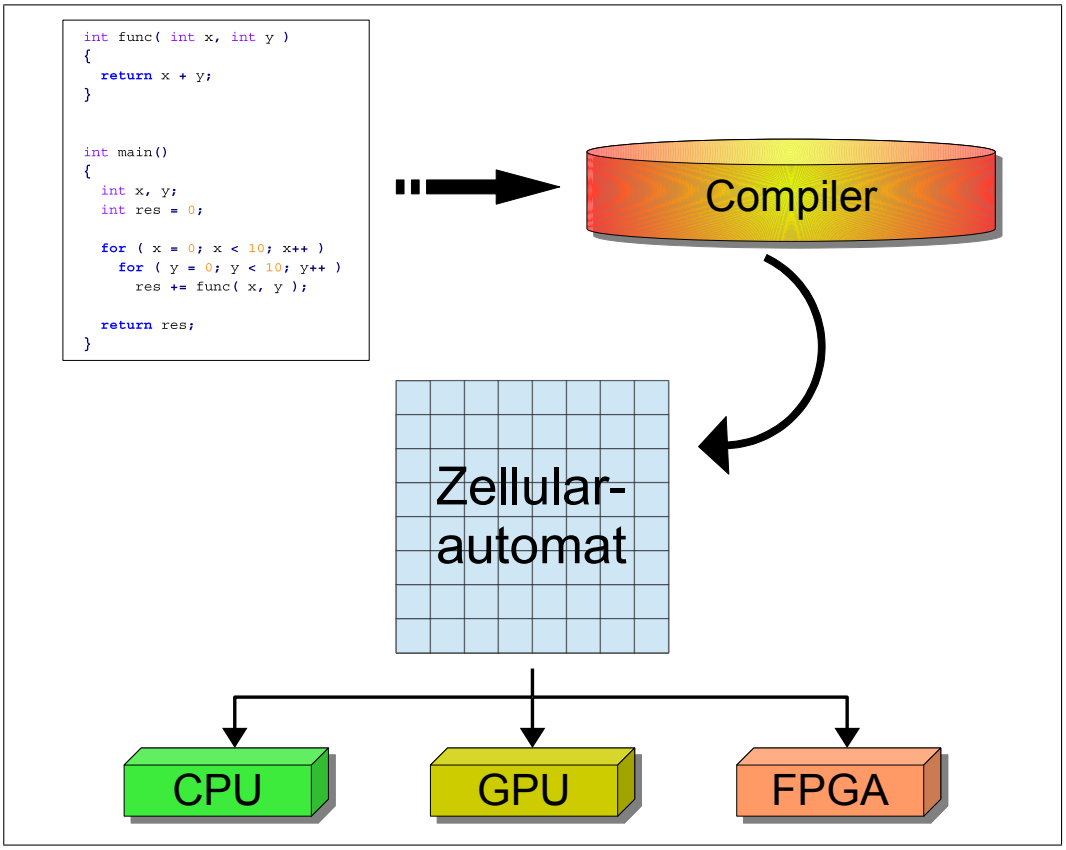


Abbildung 1.1: Überblick über das entwickelte System

rautomaten ist Gegenstand von Kapitel 4.

Kapitel 5 beinhaltet die Ergebnisse experimenteller Untersuchungen an verschiedenen Algorithmen und daraus generierten Zellularautomaten sowie Vergleiche zwischen den verschiedenen Laufzeitumgebungen zur Auswertung der Zellularautomaten.

Mit Kapitel 6 folgt eine abschließende Zusammenfassung über die vorliegende Arbeit und in Kapitel 7 ein Überblick über mögliche Weiterentwicklungen des vorgestellten Systems.

Einige Aspekte und Ergebnisse dieser Arbeit wurden bereits in [27] veröffentlicht.

Alle im Text oder in Abbildungen verwendeten Marken-, Produkt- und Firmennamen sind Eigentum der jeweiligen Inhaber bzw. Hersteller.

2 Grundlagen

Dieses Kapitel beschreibt grundlegende und für die Arbeit relevante Verfahren, gibt einen Überblick über den Stand der Technik und klärt in den anschließenden Kapiteln verwendete Begriffe.

2.1 Parallelisierungstechniken

Der erste Teil des Kapitels befasst sich mit existierenden Parallelisierungstechniken und stellt verschiedene Forschungsprojekte für unterschiedliche Zielsysteme vor.

2.1.1 Die historischen Anfänge der Parallelisierung

Nachdem Konrad Zuse 1941 den ersten ersten universell programmierbaren Rechner der Welt, den *Z3* [117][119], vorgestellt hat und 1945 die *Von-Neumann-Architektur* in einem Technischen Bericht [108] vorgestellt wurde, sind weltweit mehr und mehr Computer nach diesen Prinzipien entwickelt worden.

Bereits wenige Jahre nachdem die ersten Computer auf dem Markt erschienen, folgten bereits die ersten Parallelisierungstechniken. Eine Möglichkeit zur gleichzeitigen Verarbeitung von Daten, die 1961 im *IBM 7030 Stretch* Computer [17] und 1962 im *ILLIAC II* Computer [16] der Universität von Illinois eingesetzt wurde, ist das *Pipelining* von Instruktionen. Anstatt einen Befehl komplett in einem Takt abzuarbeiten, werden Befehle in mehrere kleine Aufgaben aufgeteilt, beispielsweise beim fünfstufigen Pipelining in:

1. Laden des Befehlscodes,
2. Dekodieren des Befehls,
3. Laden der Daten,
4. Ausführung des Befehls,
5. Zurückschreiben des Ergebnisses.

Durch die Einfachheit der Teilaufgaben, verglichen mit der gesamten Aufgabe, ist ein höherer Takt zur Bearbeitung möglich. Gleichzeitig kann bei Abschluss der ersten Phase sofort ein neuer Befehl in die Pipeline geladen werden. Die Hardware ermöglicht somit parallele Verarbeitung auf Instruktionsebene. Programmierer müssen davon keine Kenntnis haben, da sich das System nach „außen“ verhält, wie eine Architektur ohne Pipelining.

1964 erschien der von der Firma *Control Data Corporation* entwickelte *CDC 6600* [103]. Es war der erste Computer, der als Supercomputer bezeichnet wurde, da er circa 10-mal schneller rechnete als alle anderen Computer zu dieser Zeit [21]. Eine Neuerung war seine *superskalare* CPU Architektur. Dabei handelt es sich um eine Weiterentwicklung der Parallelität auf Instruktionsebene. Neben der CPU hat der *CDC 6600* zehn funktionale Einheiten, die unabhängige mathematische Operationen parallel bearbeiten können. Der *CDC 6600* zählt zu einem der ersten Vertreter der *RISC (Reduced Instruction Set Computer)* Architektur [85]. Die CPU hat einen sehr reduzierten und damit schnell zu verarbeitenden Befehlssatz; die eigentliche Berechnung übernehmen die funktionalen Einheiten. Im Gegensatz dazu werden Computer mit klassischem (komplexen) Befehlssatz der *CISC (Complex Instruction Set Computer)* Architektur zugeordnet.

Mit der Verdrängung der Vakuumröhren durch Transistoren wurden die Computer Mitte der 1960er Jahre zunehmend leistungsfähiger, energiesparender und ausfallsicherer. Zu etwa derselben Zeit formulierte *Gordon Earle Moore*, einer der Mitbegründer der *Intel Coporation*, seine Vermutung, dass sich die Anzahl der Transistoren in einem integrierten Schaltkreis etwa alle ein bis zwei Jahre verdoppeln wird [78]. Diese Behauptung, später als *Mooresches Gesetz* bezeichnet, hat nach Aussage der *Intel Coporation* auch heute noch ihre Gültigkeit [56].

Eine größere Integrationsdichte von Transistoren kann beispielsweise genutzt werden, um die Leistungsfähigkeit von Prozessoren zu erhöhen. Kleinere Transistoren haben eine geringere Leistungsaufnahme, wodurch eine Erhöhung des Taktes und somit eine höhere Verarbeitungsgeschwindigkeit möglich ist. Das *Mooresche Gesetz* und die geschichtliche Entwicklung der Anzahl der Transistoren auf einem Chip wird in Abbildung 2.1 am Beispiel der von *Intel* entwickelten Prozessoren von 1970 bis 2005 dargestellt.

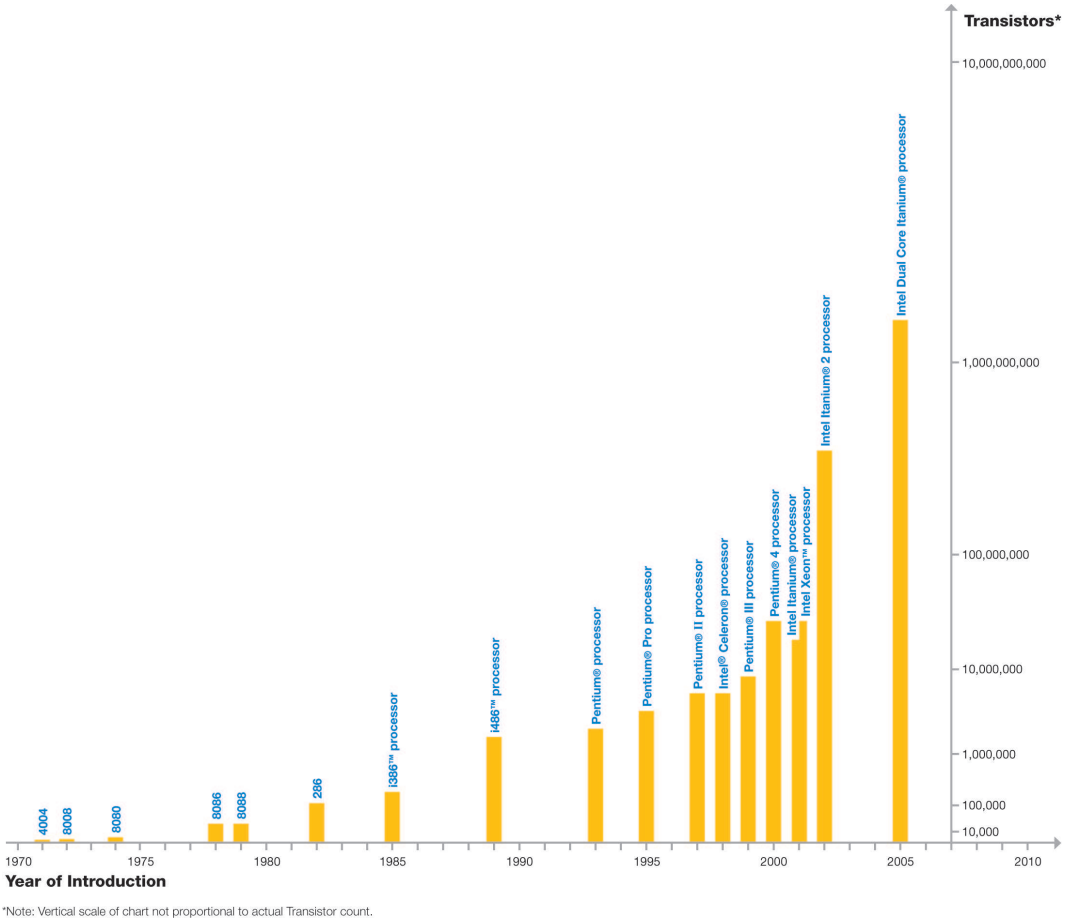


Abbildung 2.1: Das *Moorsche Gesetz* am Beispiel der Intel Prozessoren von 1970 bis 2005 (Quelle: intel.com)

Die bisher vorgestellten Methoden zur parallelen Ausführung beschränken sich auf die Nutzung von Parallelität auf Befehlsebene. Die zu verarbeitenden Programmteile müssen sehr oft kommunizieren und es werden viele Synchronisationspunkte benötigt. Sie sind somit *feingranularen* Parallelitätsansätzen zuzuordnen [96]. Mitte der 1960er Jahre erkannte man, dass es durch geringfügige Änderungen der Hardware möglich ist, mehrere vollwertige Prozessoren in einen Computer zu integrieren. Durch die Entwicklung solcher *Multiprozessorsysteme* war eine kostengünstige (im Vergleich zu schnelleren Prozessoren oder mehreren Computern) Steigerung der Geschwindigkeit und Berechenbarkeit von Algorithmen möglich. Da die zu verarbeitenden Daten zur Ausführung der Programme aufgeteilt werden und bei entsprechender Programmierung nur

wenige Synchronisationspunkte nötig sind, zählt man diese Parallelisierungstechniken zu den *grobgranularen* Ansätzen [96]. Ein Beispiel hierfür ist das 1967 von IBM entwickelte *IBM System/360 Model 67* [53].

Nach *Flynnscher Klassifikation* [30] handelt es sich bei allen bisher vorgestellten Architekturen entweder um *SISD (Single Instruction, Single Data)* Rechner, die einen Befehl auf einen Eingabestrom anwenden oder *MIMD (Multiple Instruction, Multiple Data)* Rechner, die mehrere Prozessoren/Einheiten nutzen, um unterschiedliche Befehle auf unterschiedliche Daten anzuwenden. In den 1970er Jahren wurde eine weitere hardwareseitige Parallelisierungstechnik eingeführt, die so genannten *Vektor-* bzw. *Arrayprozessoren*. Sie bilden eine zusätzliche Klasse in der *Flynnschen Klassifikation*. Vektorprozessoren gehören zu den *SIMD (Single Instruction, Multiple Data)* Rechnern. Bei ihnen wird ein Befehl auf mehrere Daten gleichzeitig angewendet, zum Beispiel bei Matrizenoperationen. Bekannte Supercomputer mit Vektorprozessoren aus den 1970er Jahren sind zum Beispiel das *Illiac IV System* [14] und der *CRAY-1* [91].

Ein von der *Von-Neumann-Architektur* abweichendes Rechnermodell stellen die Mitte der 1970er entwickelten *Datenflussrechner* dar [106]. Bei dieser Architektur verzichtet man auf die explizite Steuerung durch Kontrollflusselemente. Die Programme werden implizit über ihre Datenabhängigkeiten gesteuert und es entsteht ein asynchrones bzw. selbstsynchronisierendes System, dass auf mehrere Prozessoren aufgeteilt werden kann und unterschiedliche Programmteile parallel bearbeitet.

Neben der Integration vieler und immer schnellerer Prozessoren sowie der Anwendung verschiedener Parallelisierungstechniken in Supercomputern wurden auch spezielle parallele Programmiersprachen für *Verteilte Systeme* entwickelt [9]. Sie ermöglichen es, Systeme mit getrennten Speichern zusammenzuschließen und sie gemeinsam an einem Problem/Programm arbeiten zu lassen. Dadurch können geographisch entfernte Rechner als zusätzliche Ressource für komplexe Algorithmen genutzt werden.

Ein unter wirtschaftlichen Aspekten kritisch zu betrachtender Punkt bei der Verwendung einer großen Anzahl an Prozessoren/Ressourcen, ist der maximal zu erreichende Anstieg der Performance. Die maximale Beschleunigung eines Programms durch Parallelisierung ist beschränkt durch das *Amdahlsche Gesetz* [3]. Es besagt, dass der sequentielle Anteil eines Problems ausschlaggebend für den möglichen Beschleunigungsfaktor durch Parallelisierung ist. Woraus abzu-

leiten ist, das eine obere Grenze für den maximalen Performancezuwachs durch Parallelisierung, egal wie viele Prozessoren/Ressourcen man zur Verfügung hat, existiert.

$$S(n) = \frac{1}{r_s + \frac{r_p}{n}} \tag{2.1}$$

Formel 2.1 beschreibt das *Amdahlsche Gesetz* und den theoretisch zu erreichenden maximalen Beschleunigungsfaktor durch die Parallelisierung eines Programms. Parameter n ist die Anzahl der Prozessoren, r_s der sequentielle Anteil des Programms und r_p der parallel ausführbare Anteil. Weiterhin gilt für die Summe des sequentiellen und parallelen Anteils: $r_s + r_p = 1$.

Mit steigender Zahl an Prozessoren, steigt der Einfluss des sequentiellen Anteils auf den Beschleunigungsfaktor. Geht die Anzahl der Prozessoren gegen unendlich, dann ergibt sich:

$$\lim_{n \to \infty} S(n) = \frac{1}{r_s} \ .$$

Die Anzahl der Prozessoren ist somit irrelevant, der Beschleunigungsfaktor ist allein vom sequentiellen Anteil des Programms abhängig.

Ein wichtiger Faktor für eine möglichst effiziente Ausnutzung aller zur Verfügung stehender Prozessoren ist die Aufteilung des Programms, so dass alle Prozessoren effektiv genutzt werden. Je mehr Prozessoren ausgelastet werden sollen, desto höher ist die Komplexität der Algorithmen, da sequentielle Anteile und Datenabhängigkeiten möglichst reduziert werden müssen. Das führt jedoch zu steigenden Entwicklungskosten für Software. In den 1980er Jahren waren die Entwicklungskosten bei *Cray Research, Inc.* für Software so stark gestiegen, dass sie auf einer Höhe zu den Entwicklungskosten für Hardware lagen [74].

Eine simple Aufteilung der Daten und Berechnungen, wie zum Beispiel bei Matrizenoperationen, auf alle zur Verfügung stehenden Prozessoren ist bei komplexeren Problemen mit Daten- bzw. Ressourcenabhängigkeiten nicht immer realisierbar. Aus diesem Grund entstanden verschiedene Forschungsprojekten im Bereich automatisch parallelisierender Compiler.

2.1.2 Automatisch parallelisierende Compiler für Multi-Core CPUs

Die softwareseitigen Parallelisierungstechniken bis zum Ende der 1980er Jahre waren stark auf bestimmte Supercomputer und lokale Optimierungen konzentriert [83]. Ein Beispiel für solche Verfahren ist die Umwandlung von Schleifen in *SIMD* Anweisungen für Vektorprozessoren [2], um einen möglichst hohen Datendurchsatz zu erreichen. Anfang der 1990er Jahre fokussierte man sich neben diesen klassischen Ansätzen zur Parallelisierung zusätzlich auf universelle/globale Optimierungen zur Verbesserung der Parallelität.

Eines der ersten großen Projekte, das eine solche Technik zur automatischen Parallelisierung von Programmen für Mehrprozessorsysteme mit gemeinsamen Speicher einsetzt, ist das *PIPS Projekt*. Es wurde erstmals 1991 in [58] beschrieben. Ursprünglich wurde es ins Leben gerufen, um wissenschaftliche Programme zu beschleunigen.

Der *PIPS Compiler* ist nicht auf eine bestimmte Zielarchitektur beschränkt und kann funktionsübergreifende (globale) Optimierungen durchführen. Es ist ein *Source-to-Source Compiler*, dass heißt er erzeugt aus Quellcode wieder Quellcode derselben oder einer anderen Programmiersprache. Der *PIPS Compiler* war anfangs, wie die meisten Compiler für wissenschaftliche Berechnungen Anfang der 1990er Jahre, auf Fortran [8] als Programmiersprache beschränkt.

Nach und nach kamen weitere Funktionen hinzu. Mittlerweile unterstützt der Compiler außer Fortran auch C als Programmiersprache. Eine Übersicht über das Projekt und den aktuellen Entwicklungsstand ist in [87] zu finden.

Ein anderes Ziel verfolgt das von der *NASA* unterstützte *ParaScope Projekt* [23]. Dabei handelt es sich um eine Entwicklungsumgebung, die mehrere Tools miteinander verbindet, um die Softwareentwicklung für Mehrprozessorsysteme zu vereinfachen. Als Programmiersprache wird bei diesem Projekt ebenfalls Fortran verwendet. Neben einem Tool zur Datenabhängigkeitsanalyse enthält es einen Editor zur Entwicklung paralleler Algorithmen und einen Debugger für parallele Anwendungen.

Hauptziel dieses Projektes ist es nicht, eine vollautomatische Umsetzung von sequentiellen Programmen auf parallele Programme zu liefern. Vielmehr soll das System Programmierer bei der Entwicklung paralleler Programme visuell

unterstützen sowie teilweise automatische Optimierungen am Code durchzuführen. Die Programme sollen zudem zur Laufzeit überprüft werden können (Debugging), was bei paralleler Abarbeitung deutlich komplexer als bei sequentieller Abarbeitung ist.

Ein weiteres Projekt, welches sich mit der automatischen Parallelisierung von Fortran-Code beschäftigt, ist in [11] beschrieben. Das Hauptziel dieses Projektes, genannt *Polaris*, liegt in der Optimierung verhältnismäßig großer Programme zur Bearbeitung beliebiger Probleme. Laut Aussage der Autoren soll ihr Compiler, im Gegensatz zu den meisten anderen Compilern für Mehrprozessorsysteme zu dieser Zeit, unter Realbedingungen performante Ergebnisse liefern. Die Programme müssen nicht speziell für den Compiler designet werden und können einen beliebigen (Code-)Umfang haben.

Neben der automatischen Parallelisierung wird in [11] ein dynamisches Verfahren zur Ermittlung von Parallelisierungskapazitäten zur Laufzeit von Programmen beschrieben.

Der an der Universität von Stanford entwickelte *SUIF Compiler* [40] ist, wie das *PIPS Projekt*, nicht auf eine bestimmte Zielarchitektur festgelegt. Ziel ist es vielmehr einen optimierten Zwischencode zu generieren, der die Parallelisierungskapazitäten eines sequentiellen Programms beschreibt. Das Projekt ist nicht auf Supercomputer ausgerichtet, sondern soll mittels grobgranularer Parallelität beliebige Anwendungen für Computer mit mehreren Prozessoren und gemeinsamem Speicher beschleunigen.

Der *SUIF Compiler* ist in der Lage globale Datenabhängigkeiten zu berücksichtigen und hat keine Beschränkungen bezüglich der Größe der von ihm bearbeitbaren Programme. Ist der Code ungeeignet für Parallelisierungen nimmt der Compiler automatisch Codetransformationen vor. Zum Beispiel ersetzt er globale Arrays durch private Kopien für jeden Prozessor, um Fehler bei Lese- und Schreibzugriffen zu verhindern. Neben den Techniken zur Parallelisierung nimmt der Compiler Optimierungen des Speichermanagements vor, so dass jeder Prozessor so oft wie möglich auf seinen Cache zurückgreifen kann. Dadurch wird der Flaschenhals des langsamen Zugriffs auf den globalen Arbeitsspeicher weitestgehend reduziert.

Einige andere Forschungsprojekte nutzen den vom *SUIF Compiler* erzeugten Zwischencode in Form von *abstrakten Syntaxbäumen* (siehe Kapitel 3.2). Neben

statischen und dynamischen Analyseverfahren zur Steigerung der Softwarezuverlässigkeit, zum Beispiel in [42] oder in [45], sowie Verfahren zur Pointer Alias Analyse [92] basieren weitere Forschungsprojekte im Bereich der Parallelisierung auf dem vom *SUIF Compiler* erzeugten Zwischencode. Ein Beispiel hierfür ist das *SUIF Explorer Projekt* [72], bei dem der Benutzer visuell auf Parallelisierungskapazitäten hingewiesen wird.

Beim *Machine SUIF Compiler* [95] handelt es sich um ein Tool, das ebenfalls auf dem *SUIF Compiler* basiert. Es erzeugt aus den maschinenunabhängigen abstrakten Syntaxbäumen des *SUIF Compilers* einen architekturspezifischen optimierten Code. In [49] wird beschrieben, wie sich dieser optimierte Code in einen Kontrollflussgraphen des Programms sowie zu den Basisblöcken zugehörige Befehlslisten zur Weiterverarbeitung umwandeln lässt.

Aufbauend auf den beschriebenen Verfahren wird in [50] eine Erweiterungsbibliothek des *Machine SUIF Compilers* vorgestellt, mit deren Hilfe sich der Kontrollflussgraph in eine *Static-Single-Assignment-Darstellung (SSA)* [6][90] transformieren lässt. In dieser Darstellung wird jeder Variable exakt einmal ein statischer Wert zugewiesen. Das führt zu einer expliziten Beschreibung von Datenabhängigkeiten und kann für Optimierungsverfahren, wie zum Beispiel für *Dead-Code-Elimination* (siehe Kapitel 3.3.3), genutzt werden. Weitere Informationen zur Generierung und Beispiele zur *SSA-Darstellung* werden in Kapitel 3.4.1 behandelt.

Ein weiteres Projekt, welches auf dem *SUIF Compiler Framework* basiert ist das *PEGASUS Projekt* [18]. Als Eingabesprache wird C verwendet und der Compiler produziert daraus eine maschinenunabhängige Zwischensprache für verschiedene Analysen und Optimierungen. Ziel des Projektes ist eine effiziente Speicherung des Daten- und Kontrollflusses sowie die Speicherung der Synchronisationspunkte für Operationen mit Seiteneffekten.

Aus den Kontrollflussgraphen des *SUIF Compilers* werden zunächst *Hyperblöcke* gebildet. Hyperblöcke sind gerichtete azyklische Graphen und bilden eine (eventuell spekulative) nicht unterbrechbare Ausführungseinheit des *PEGASUS Graphen*. Sie haben genau einen Eingang und einen oder mehrere Ausgänge. Kleinste Einheit für Hyperblöcke in *PEGASUS* sind die Schleifenkörper der innersten Schleifen. Äußere Schleifen enthalten wiederum mehrere Hyperblöcke.

Nach Bildung der Hyperblöcke werden aus dem Kontrollflussgraphen die Bedingungen für Kantenübergänge innerhalb und zwischen den Hyperblöcken extrahiert. Im Gegensatz zur *SSA-Darstellung* gibt es im *PEGASUS* Framework keine ϕ-Funktionen (siehe Kapitel 3.4.1). Stattdessen werden so genannte *Multiplexor* benutzt. Alle eventuell an einer Stelle im Programm benötigten Daten und die zugehörigen Bedingungen werden an die Multiplexor gesandt. Diese wählen dann die zu verwendenden Daten anhand der Bedingungen aus. Über ein Handshake-Verfahren wird anschließend der Wert beim Versender der Daten gelöscht, so dass dieser für neue Berechnungen eingesetzt werden kann. Das ist zum Beispiel bei Schleifen und Funktionsaufrufen notwendig. Durch das beschriebene Verfahren erzeugt der Compiler aus dem gegebenen Kontrollflussgraphen einen Datenflussgraphen.

Der *PEGASUS Compiler* kann mehrere Optimierungen durchführen. Dazu zählen unter anderem klassische Verfahren wie die Entfernung von nicht erreichbarem Code oder die Ersetzung von konstanten, dass heißt zum Kompilierungszeitpunkt bekannten, Berechnungen. Außerdem wird im *PEGASUS Compiler* eine Technik zur spekulativen Ausführung von Code eingesetzt. Kantenbedingungen werden hierfür auf **true** gesetzt, wenn es keine Nebeneffekte (Veränderung bzw. Verfälschung der Ergebnisse) hat. Dies dient insbesondere der Beschleunigung bei paralleler Ausführung des Codes. Eine Optimierungstechnik, die bei der Ausführung zum Tragen kommt, ist die so genannte *lenient evaluation* [105]. Dabei wird ausgenutzt, dass bestimmte Operationen ein Ergebnis liefern können, bevor alle Operanden bekannt sind. Im *PEGASUS Compiler* wird die Technik zum Beispiel bei logischen sowie arithmetischen **UND**-Operationen, bei denen ein Operand Null und somit das Ergebnis ebenfalls Null ist, eingesetzt. Darüber hinaus ergibt sich eine Einsatzmöglichkeit für logische **ODER** Operationen, bei denen ein Operand gleich Eins und somit das Ergebnis ebenfalls gleich Eins ist. Das Verfahren verkürzt, bei Auftreten einer der Bedingungen, den *kritischen Pfad* des Programms und reduziert somit den sequentiellen Anteil des Codes (siehe *Amdahlsches Gesetz* in Kapitel 2.1.1).

In [101] wird ein Verfahren zur Nutzung grobgranularer Parallelität für Streaming-Anwendungen vorgestellt. Ziel dieser Technik ist es, verschiedene Audio- und Videobearbeitungsalgorithmen sowie Programme zur digitalen Si-

gnalverarbeitung zu beschleunigen. Eine Besonderheit des Verfahrens besteht in der Konzentration auf grobgranulares Pipelining anstatt auf Daten- oder Aufgabenparallelität.

Es handelt sich um eine halbautomatische Technik, die vom Programmierer unterstützt werden muss. Der Programmierer legt anhand von bestimmten *defines*, in den zu parallelisierenden Schleifen, die einzelnen Stufen des Pipelinings fest. Danach erstellt das Tool während einer Test-Ausführung des Programms einen so genannten *Stream-Graphen*, der die Kommunikation aller Pipeline-Stufen enthält. Ist der Programmierer mit dem Grad der Parallelität in dem Graphen zufrieden, kann er den Code zusammen mit den durch das Tool generierten Makros erneut übersetzen und erhält das finale (parallelisierte) Programm.

Das *PLuTo Projekt* [12][13] beschäftigt sich mit der automatischen Parallelisierung von geschachtelten Schleifen. Die dazu verwendete Technik basiert auf dem *Polytopmodell*. Dabei handelt es sich um ein mathematisches Modell, bei dem Schleifen und Variablenabhängigkeiten durch lineare Gleichungssysteme in Matrizenform dargestellt werden. Diese Darstellung ermöglicht mathematisch korrekte Transformationen, um maximale Parallelität aus gegebenen Algorithmen zu extrahieren. Anschließend wird das lineare Gleichungssystem wieder in einen Quellcode umgewandelt, der dann zu einem Programm mit paralleler Schleifenausführung compiliert werden kann.

Der *PLuTo Compiler* ist Teil der *Polyhedral Compiler Collection (PoCC)* [88], einer Ansammlung verschiedener Tools zur Optimierung von Algorithmen, basierend auf dem Polytopmodell.

Automatische Parallelisierung von Schleifen wird, neben den bereits beschriebenen Forschungsprojekten, auch von einigen Standard-Compilern für C/C++-Code direkt unterstützt, zum Beispiel vom *GCC Compiler* seit Version 4.4 [33], vom *Microsoft Visual Studio* ab Version 2012 [76] sowie vom *Intel C++ Compiler* seit 2011 [54].

2.1.3 Übersetzung sequentieller Programme auf Many-Core GPUs

Neben den Arbeiten zur automatischen Parallelisierung für Multi-Core CPU Systeme gibt es Forschungen zur Nutzung der Many-Core Prozessoren von Gra-

fikkarten. Die Gemeinsamkeit der Forschungsprojekte für Grafikkarten liegt in der Nutzung bekannter Techniken zur Parallelisierung von CPU-Programmen.

Zusätzlich zu den Schwierigkeiten mit Datenabhängigkeiten müssen bei der Übersetzung von Programmen für GPUs weitere Probleme gelöst werden. Dazu zählen unter anderem die Erstellung von Synchronisationspunkten zur Kommunikation mit der CPU sowie die Einführung notwendiger Kopieroperationen für Daten. Die Vielzahl an zur Verfügung stehenden Rechenkernen muss zudem möglichst vollständig ausgenutzt werden, um ein effizientes Programm zu erhalten.

Das *Par4All Projekt* [4] ist eine Erweiterung des in Kapitel 2.1.2 beschriebenen *PIPS Compilers*. Anstatt C-Code mit Anweisungen zur Parallelisierung auf der CPU zu erzeugen, generiert der Compiler *OpenCL*-Code, der bestimmte Teile der Anwendung auf die GPU abbildet. Darüber hinaus werden die für die getrennten Adressräume von CPU und GPU nötigen Kopieroperationen der Daten sowie die Aufruf- und Steuerbefehle für den CPU-Teil des Programms im Quellcode eingefügt.

Ein vergleichbares System, welches ebenfalls das Polytopmodell benutzt, um Parallelität in Schleifen zu extrahieren und auf die GPU abzubilden ist in [107] beschrieben. Bei diesem Projekt werden neben den zuvor erläuterten Techniken lokale Speicheroptimierungen durchgeführt, um den gemeinsamen Speicher der GPU Threads möglichst optimal zu nutzen.

Zusätzlich zur Parallelisierung von Schleifen wird in [60] automatisches Pipelining zur Verbesserung der Performance verwendet. Außerdem soll der Compiler des Projektes die Kommunikation zwischen CPU und GPU optimieren bzw. minimieren.

Während die bisher genannten Verfahren sich hauptsächlich mit grobgranularer Aufgabenparallelität und Pipelining beschäftigen, wird bei dem in [10] beschriebenen Projekt ein anderes Ziel verfolgt. Das so genannte *KPN2GPU Projekt* konzentriert sich auf die Analyse von Datenabhängigkeiten in Schleifen, um feingranulare Parallelität auf Datenebene zu ermöglichen.

```
entity Adder is
  Port ( x : in  integer
           range 0 to 15;
         y : in  integer
           range 0 to 15;
         z : out integer
           range 0 to 15 );
end Adder;

architecture Behavioral
  of Adder is
begin

  z <= x + y;

end Behavioral;
```

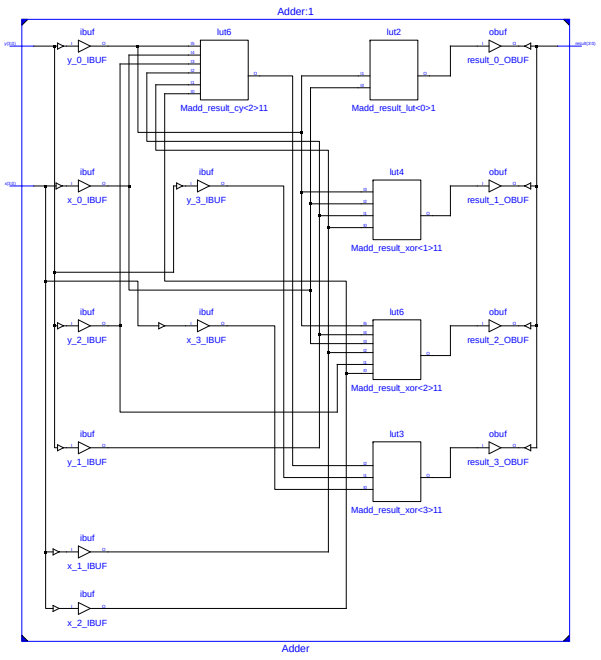


Abbildung 2.2: 4-Bit Addierer in VHDL und die zugehörige synthetisierte Schaltung

2.1.4 High-Level Design Tools für Hardware

Wie in Kapitel 1 beschrieben, bieten FPGAs eine kostengünstige und performante Möglichkeit parallele Algorithmen umzusetzen. Ein großes Problem für Softwareentwickler stellt allerdings ihr Programmiermodell dar. Eine Möglichkeit zur Programmierung ist der direkte Entwurf der logischen Schaltungen in Form von Schaltplänen. Das ist jedoch nur bei sehr kleinen Schaltungen realisierbar. FPGAs werden daher meist mithilfe einer Hardwarebeschreibungssprache programmiert. Beispiele hierfür sind *VHDL (Very High Speed Integrated Circuit Hardware Description Language)* und *Verilog*. Mittels dieser Sprachen beschreibt/programmiert der Entwickler die zu erstellende Hardware. Anschließend generiert ein *Synthesetool* aus der Hardwarebeschreibung eine logische Schaltung. Durch vorhandene Bibliotheken und vordefinierte Funktionen wird die Arbeit für Entwickler deutlich erleichtert. Abbildung 2.2 zeigt einen Anwendungsfall von VHDL als Hardwarebeschreibungssprache und die dazugehörige synthetisierte Schaltung.

Neben VHDL und Verilog gibt es auch andere Sprachen, die ebenfalls zur Programmierung oder Simulation von FPGAs eingesetzt werden können. Die nachfolgend beschriebenen Programmiersprachen haben als Gemeinsamkeit eine auf C basierende Syntax und unterstützen einen Teil der aus C bekannten Funktionen. Ein Compiler erzeugt aus dem Code nach Fertigstellung des Programms wiederum eine Hardwarebeschreibung, die letztendlich auf einen FPGA abgebildet werden kann.

Eine erstes Beispiel für eine solche Hochsprache ist *Handel-C* [7]. Ursprünglich an der Universität von Oxford entwickelt, ist *Handel-C* seit 2009 Teil der *DK Design Suite* von *Mentor Graphics*. Eine Besonderheit der Sprache sind Codewörter zur Definition paralleler Bereiche im Programm sowie Kanäle zur Kommunikation zwischen parallelen Threads.

FpgaC ist ein Open-Source Projekt. Es stellt eine Weiterentwicklung des 1995 vorgestellten *Transmogrifier C Tools* [34] der Universität von Toronto dar. Neben einer Teilmenge der C-Syntax werden spezielle Definitionen benutzt, um Eingabe- bzw. Ausgabepins anzusteuern.

SystemC unterstützt neben C auch eine Teilmenge der C++-Syntax, so dass mit Klassen und Methoden gearbeitet werden kann. *SystemC* ist von der IEEE als Standard genehmigt [52] und wird von vielen Herstellern unterstützt. In *SystemC* werden, ähnlich zu *Handel-C*, Kanäle zur Kommunikation paralleler Threads eingesetzt. Eine Besonderheit ist die eingebettete Simulationsumgebung, die *SystemC* interessant für schnelles Entwickeln und Testen macht. Im Gegensatz zur Hardwaresimulation auf Logikebene kann diese deutlich schneller ausgeführt werden. Dadurch sind Änderungen am Design sowie das Überprüfen einer digitalen Schaltung auf Korrektheit effizient möglich.

Der Vorteil in der Nutzung C-basierter Hochsprachen zum Designen von Hardware liegt in der für Softwareentwickler bekannten Syntax. Die genannten Programmiersprachen haben allerdings den Nachteil, dass sie trotz einer C/C++ ähnlichen Sprachstruktur Kenntnisse des Entwicklers über die entstehende Schaltung und Parallelität voraussetzen. Module müssen programmiert und entsprechend synchronisiert werden. Es ist zudem notwendig, parallel zu bearbeitende Abschnitte der Algorithmen explizit anzugeben und Ein- und Ausgaben über Pins bzw. Ports zu koordinieren.

Einen anderen Weg beschreitet das *CASH Projekt* [19]. Dabei handelt es

sich um einen Compiler zur automatischen Übersetzung von ANSI C-Code auf Hardware. Es wird allerdings, anders als in den bisher vorgestellten Projekten, vom Programmierer keinerlei Hardwarewissen benötigt. Das Framework basiert auf dem in Kapitel 2.1.2 vorgestellten *PEGASUS Compiler* und nutzt dessen generierten Zwischencode in Form von Datenflussgraphen.

Jeder Knoten im *PEGASUS*-Datenflussgraphen wird vom Compiler in eine einzelne Hardwarestruktur abgebildet. Zur Synchronisation und zum Datenaustausch der einzelnen Knoten wird ein Handshake-Verfahren über einen Kommunikationskanal benutzt. Sobald versendete Daten beim Empfänger ankommen schickt dieser eine Empfangsbestätigung und der Sender kann die Daten löschen. Somit ist keine globale Steuerung und Kontrolle notwendig. Die generierte Hardware ist selbstsynchronisierend und kann auf *Anwendungsspezifische integrierte Schaltungen (ASICs)* abgebildet werden. Auf eine Programmierung von FPGAs haben die Entwickler verzichtet, da nach ihrer Aussage FPGAs ungeeignet bzw. ineffizient bei einer taktfreien asynchronen Verarbeitung sind.

Problematisch bei dem vorgestellten Verfahren sind Speicherzugriffe auf Variablen, die als globaler Speicher implementiert sind. Spezielle *Load/Store*-Knoten und Warteschlangen, die die Reihenfolgen festlegen, steuern hierbei den Zugriff. Dies ist gleichzeitig der Flaschenhals des Systems und macht einen Großteil der Verarbeitungszeit der Algorithmen aus.

Vergleichbar ist das *CASH Projekt* mit einem dynamisch erzeugten Datenflussrechner (siehe Kapitel 2.1.1), bei dem im Falle des *CASH Frameworks* die Hardware nicht vorgegeben ist, sondern angepasst an die umzusetzenden Algorithmen erzeugt wird.

Die Parallelisierung von Streaming-Applikationen, zum Beispiel zur Bearbeitung von Video- oder Audiosignalen, ist das Ziel des *Daedalus Frameworks*. Der in [102] beschriebene Compiler kann bestimmte C-Algorithmen auf *Kahn Process Networks* [36] abbilden. Dabei handelt es sich um ein selbstsynchronisierendes Netz aus untereinander kommunizierenden Blöcken. Es wird, wie beim *CASH Framework*, ein Datenflussmodell zur Ausführung und Steuerung des Programms benutzt. Der Hauptunterschied zum *CASH Framework* besteht darin, dass keine Hardwareelemente aus dem Datenflussgraphen gebildet werden. Stattdessen wird das Datenflussmodell unterteilt und in Software für

synthetisierte Prozessoren abgebildet, die das *Kahn Process Network* repräsentieren.

Auf der gleichen Technik, die das *Daedalus Framework* verwendet, basiert das *Compaan Tool* [66]. Das mittlerweile kommerzielle Programm konzentriert sich nicht nur auf die Abbildung auf synthetisierte Hardware, sondern kann auch parallele Codes für Multi-Core CPUs und Grafikkarten erzeugen. Ziel des *Compaan Tools* ist es, performancekritische Bereiche eines Streaming-Programms zu identifizieren und diese Teile durch den Einsatz von paralleler Software/Hardware zu beschleunigen.

In [112] werden aus der Softwareentwicklung bekannte Vektorisierungstechniken auf Hardware übertragen. Die Vektorisierung wird dabei auf die innersten Schleifenkörper in dem zu übersetzenden Programm angewandt. Im Unterschied zur Softwareimplementierung werden hierbei keine explizite Vektorbefehle eingeführt. Vielmehr werden alle Instruktionen des Schleifenkörpers vektorisiert und der Datenfluss ausgenutzt, um ein Pipelining der Daten zu ermöglichen. Zudem nimmt der Compiler verschiedene Schleifenoptimierungen vor, um die Parallelität der Algorithmen zu erhöhen.

Eine Besonderheit des Compilers sind zwei verschiedene Bearbeitungsmodi. Im ersten Modus, dem so genannten Hardware-Modus, wird das gesamte Programm auf entsprechende Hardware abgebildet. Dieser Modus kann nur für vollständig synthetisierbare Programme benutzt werden. Ausnahmen davon stellen zum Beispiel rekursive Funktionen oder Bibliotheksfunktionen dar. In einem zweiten möglichen Modus, dem Codesign-Modus, werden nicht synthetisierbare Teile in einen Mikroprozessor ausgelagert. Der Rest des Programms wird in Hardware realisiert, um ein möglichst performantes System zu erzeugen. Die benötigten Kommunikationsschnittstellen zwischen der Hardware und der im Mikroprozessor implementierten Software werden vom Compiler automatisch erzeugt.

Das *LegUp Projekt*, dass sich mit der Erzeugung einer hybriden Hardware/ Software-Architektur aus regulärem C-Code beschäftigt, wird in [20] vorgestellt. Das aus dem Code kompilierte System besteht aus einer Software für einen MIPS Softcore und in Hardware ausgelagerten Beschleunigerfunktionen. Die Hardware und Software kommunizieren über ein automatisch erzeugtes Bussystem.

Die Aufteilung des Programms in Hardware/Software wird nicht statisch anhand der Datenabhängigkeiten erzeugt. Vielmehr wird das Programm zunächst in Software für den MIPS Prozessor umgesetzt und durch Profiling analysiert. Anschließend werden, für eine Hardwareimplementierung geeignete, performancekritische Bereiche des Programms vom Compiler ausgewählt und in Hardware abgebildet sowie die Kommunikationsschnittstellen zwischen der Hardware und der Software erzeugt.

Eine Übersicht über weitere existierende Verfahren bzw. Forschungen zur High-Level-Entwicklung (re-)konfigurierbarer Systeme ist in [64] zu finden.

2.2 Zellularautomaten

Die bisher vorgestellten Parallelisierungstechniken haben ein klassisches Berechnungsmodell mit genauen Vorgaben für sequentielle und parallele Abschnitte sowie explizite Synchronisationspunkte als Basis. Dieser Teil des Kapitels beschäftigt sich mit einem stark davon abweichenden Berechnungsmodell, den so genannten *Zellularautomaten*.

Zellularautomaten sind selbstsynchronisierende Berechnungsmodelle, die nur wenige Regeln benötigen, um komplizierte diskrete dynamische Systeme aus unterschiedlichen Fachrichtungen, unter anderem Biologie, Chemie und Physik, zu simulieren.

Sie wurden von *John von Neumann*, der sich in [109] mit selbstreproduzierenden Automaten beschäftigt, sowie *Konrad Zuse*, der in [118] den gesamten Kosmos als Rechenmaschine beschreibt, als Berechnungsmodelle eingeführt.

2.2.1 Klassische Zellularautomaten und das „Spiel des Lebens“

Klassische Zellularautomaten bestehen aus einem regulären n-dimensionalen Gitter von untereinander verbundenen Zellen, zugeordneten charakteristischen Zellzuständen und einer Menge von Regeln. Diese Regeln legen fest, wie der nächste Zustand einer Zelle, basierend auf dem aktuellen Zustand der Zelle sowie den Zuständen der lokalen Nachbarzellen, berechnet wird. Der Übergang von einem Zustand zum nächsten passiert bei einem synchronen Zellularautomaten für alle Zellen des Automaten gleichzeitig, das heißt beim Wechsel vom Zeitpunkt t zum Zeitpunkt $t+1$. Bei asynchronen Zellularautomaten wechseln

die Zellen zeitlich unabhängig voneinander ihre Zustände.

Der Vorteil von Zellularautomaten als Berechnungsmodell liegt in der parallelisierbaren Berechnung des nächsten Zustands einer Zelle. Die Berechnung folgt bei einem synchronen Zellularautomaten dem nachfolgenden Schema:

1. Werte die Zustände aller Zellen und ihrer Nachbarn anhand der festgelegten Regeln aus,
2. Speichere die neu berechneten Zustände in einem Zwischenspeicher,
3. Setze die neuen Zustände aller Zellen.

Der nächste Zustand einer Zelle ist lediglich von der aktuellen Zellgeneration abhängig und die neuen Zustände werden erst nach Abarbeitung aller Zellen gesetzt. Somit existieren keinerlei Datenkonflikte, da während der Berechnung nur lesend auf die Zellen zugegriffen wird. Eine Zelle kann nur ihren eigenen Zustand verändern. Dadurch sind auch beim Schreiben der neuen Zustände keinerlei Schutzmechanismen zur Verhinderung von Dateninkonsistenzen notwendig. Eine parallele Berechnung der neuen Zellzustände kann aus den genannten Gründen fehlerfrei durchgeführt werden.

Zellularautomaten sind durch die parallelisierbare Berechnung und die geringen Datenabhängigkeiten der Zellen untereinander bestens für eine effiziente Umsetzung in Software für Multi/Many-Core Architekturen und Hardwareimplementierungen geeignet. Die Leistungsfähigkeit der Umsetzung eines Zellularautomaten in einem FPGA im Vergleich zu einer Softwareimplementierung ist in [47] beschrieben.

Ein bekanntes Beispiel für klassische Zellularautomaten ist das 1970 von *John Horton Conway* entwickelte und in [32] von *Martin Gardner* beschriebene „Spiel des Lebens“ (*Game of Life*). Bei diesem zweidimensionalen Zellularautomaten gibt es genau zwei Zellzustände: „lebendig“ und „tot“. Jede Zelle hat acht Nachbarzellen, die so genannte *Moore Nachbarschaft* [77]. Die originale Regelmenge beschränkt sich auf zwei Regeln:

1. eine „lebendige“ Zelle überlebt, wenn sie **mindestens zwei** und **höchstens drei** Nachbarn hat,
2. eine „tote“ Zelle wird „lebendig“, wenn sie **genau drei** Nachbarn hat.

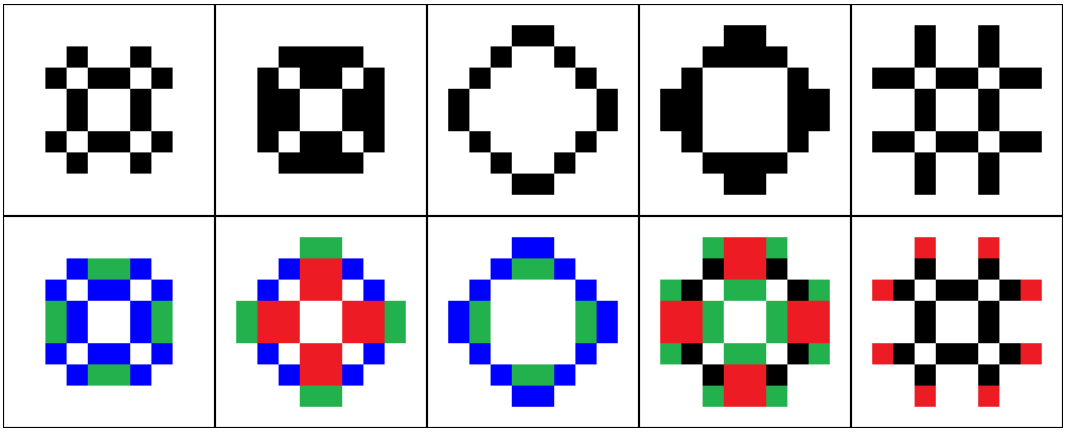


Abbildung 2.3: „Spiel des Lebens“ - Oszillierendes Objekt mit Zykluslänge fünf

Je nach Wahl der initialen Zellzustände werden unterschiedliche Muster während der Auswertung des Zellularautomaten erzeugt. So können sich ausbreitende (immer mehr „lebendige“ Zellen), stabile (Zellen verändern sich nicht) oder auch oszillierende (Muster kehrt nach einer festen Periode immer wieder) Zellularautomaten entstehen.

Ein oszillierender Zellularautomat ist in Abbildung 2.3 visualisiert. Die obere Reihe stellt die *gesamten* möglichen Zustände des Automaten in zeitlicher Reihenfolge von links nach rechts dar. In der unteren Reihe ist die Regelauswertung für das korrespondierende Bild darüber erkennbar. Blaue Zellen „überleben“ bis zum nächsten Zeitpunkt, grüne Zellen werden in der nächsten Zellgeneration „geboren“ und rote Zellen „sterben“.

2.2.2 Globale Zellularautomaten

Klassische Zellularautomaten haben den Nachteil, dass Berechnungen bzw. Simulationen, die mit ihnen durchgeführt werden können, sehr beschränkte Nachbarschaften voraussetzen. Dabei werden auf alle Zellen die gleichen Regeln angewendet.

Um einen komplexeren Algorithmus umzusetzen, wie eine *Fast Fourier Transformation (FFT)* [97], der globale Kommunikation benötigt oder lokal anzuwendende Zellregeln voraussetzt, müssen teils umständliche Codierungen vorgenommen werden. Abhängigkeiten für lokale Regeln müssen in den Zuständen der Zellen codiert werden. Dazu zählen unter anderem die Po-

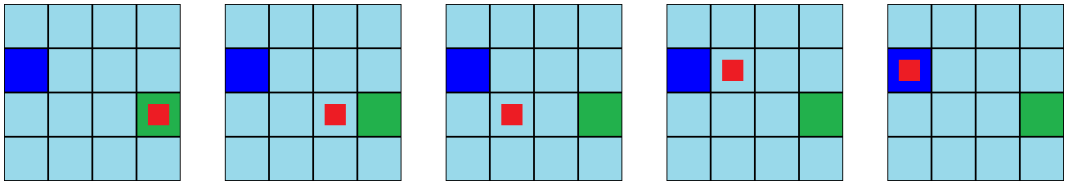


Abbildung 2.4: Weiterleitung einer Nachricht (rot) in einem klassischen Zellularautomaten vom Sender (grün) zum Empfänger (rot) über mehrere Zellen

sition der Zelle sowie bestimmte Zeitpunkte zum Ausführen von Aktionen. Kommunikation zu weiter entfernten Zellen macht es bei klassischen Zellularautomaten außerdem erforderlich, dass Nachrichten indirekt verschickt werden. Eine Nachricht wird von der sendenden Startzelle aus, über die jeweiligen Nachbarn, Schritt für Schritt weitergeschickt, bis sie ihr Ziel erreicht (siehe Abbildung 2.4). Allein für die Kommunikation werden somit mehrere Zellgenerationen benötigt, was die performante Ausführung eines Algorithmus erheblich behindert.

Eine Weiterentwicklung der Zellularautomaten für allgemeinere Probleme und Algorithmen stellen die in [46] eingeführten *globalen Zellularautomaten* dar. Sie sind nicht auf eine lokale Nachbarschaft der Zellen beschränkt. Die Zellen können in diesem Modell zu beliebigen anderen Zellen des Automaten eine Verbindung haben, unabhängig von ihrer Position im n-dimensionalen Gitter. Die Verbindung der Zellen untereinander muss nicht statisch sein; sie kann sich auf Basis der Zellregeln über den Zeitverlauf ändern. Es kann somit eine dynamisch wechselnde Nachbarschaft der Zellen vom Entwickler des Automaten realisiert und entsprechend genutzt werden.

Eine Besonderheit der globalen Zellularautomaten ist die Möglichkeit der Zuweisung unterschiedlicher Regeln an die einzelnen Zellen des Automaten. Damit sind sie nicht mehr nur für einfache Simulationen anwendbar, sondern eignen sich auch zum Lösen komplexerer Probleme. Zur effizienten Umsetzung eines Algorithmus für globale Zellularautomaten kann eine, in [48] beschriebene, experimentelle Programmiersprache namens *Global Cellular Automata Language (GCA-L)* verwendet werden. In dieser Sprache lassen sich die Zelleigenschaften und Regeln für die Zustandsänderung auf Daten- und Nachbarschaftsebene für die einzelnen Zellen beschreiben, um Algorithmen auf globale Zellularautomaten abzubilden.

2.2.3 Andere Zellularautomaten für universelle Aufgaben

Das *Trend/jTrend Projekt* [22] beschäftigt sich mit der Anwendung von Zellularautomaten für allgemeine Aufgaben. Bei dem Projekt handelt es sich um eine grafische 2D-Simulationsumgebung für Zellularautomaten inklusive einem Hochsprachencompiler zur Implementierung beliebiger Verhaltens- und Nachbarschaftsregeln.

Die Zellen des Automaten können vom Benutzer bzw. Programmierer auf einem flexiblen 2D-Gitter angeordnet und jeder Zelle eine individuelle Regel zugewiesen werden. Anschließend kann das System das Verhalten des Zellularautomaten über einen beliebigen Zeitraum simulieren.

Durch die individuelle Programmierbarkeit der Zellen im Bezug auf ihr Verhalten und ihre Nachbarbarschaft können viele Probleme auf das System abgebildet werden. Dazu zählen neben dem „Spiel des Lebens“ auch komplexere Algorithmen, beispielsweise zum Sortieren von Zahlen oder Verschlüsselungsalgorithmen.

Reconfigurable Asynchronous Logic Automata (RALA) [35] sind den asynchronen Zellularautomaten ähnlich. Der Unterschied zu Zellularautomaten besteht in der asynchronen Kommunikation und Aktualisierung der Zellen sowie in der Beschränkung der Regeln auf einfache logische Funktionen. Sie können genutzt werden, um beispielsweise Algorithmen aus der digitalen Signalverarbeitung umzusetzen. Programmiert werden sie entweder über ein direktes Platzieren von logischen Zellen und ihren Verknüpfungen oder mittels einer speziellen Hardwarebeschreibungssprache.

Weitere Forschungen zu Simulationsumgebungen, Anwendungsmöglichkeiten und Tools zum Erstellen von Zellularautomaten für allgemeine Aufgaben sind in [99] und [104] sowie [110] beschrieben. Die Gemeinsamkeit dieser und vergleichbarer Projekte liegt darin, dass sie ein Verständnis des Programmierers zum Aufbau und Programmiermodell von Zellularautomaten voraussetzen. Der Entwickler muss die Position der Zellen angeben, die individuellen Zellregeln festlegen sowie die Nachbarschaftsbeziehungen definieren.

Ein sich von den bisher aufgeführten Konzepten unterscheidendes Programmiermodell zur Nutzung von Zellularautomaten für universelle Aufgaben/Probleme wird in [79] beschrieben. Die Idee dieses Projektes ist es, Softwareent-

wicklern ein ihnen bekanntes sequentielles Programmiermodell zur Verfügung zu stellen und dieses anschließend auf ein paralleles zellulares Modell abzubilden.

Als Eingabesprache für das System dient eine experimentelle imperative Programmiersprache, genannt *Tiny Imperative Programming Language (TIP)*. Die Sprache hat eine an C bzw. Java angelehnte Syntax und ist dadurch für Softwareentwickler, die diese Sprachen beherrschen, leicht zu verstehen. Daneben beinhaltet das Projekt eine Java basierte Simulationsumgebung, die das Verhalten des aus dem TIP-Code erzeugten Zellularautomaten simulieren kann.

3 Idee und Umsetzung des Compilers und der Laufzeitumgebung

In diesem Kapitel wird der konkrete Ablauf der Umwandlung von einem sequentiellen C-Programm zu einem parallel arbeitenden Zellularautomaten erläutert.

3.1 Grundlagen und Arbeitsweise des Frameworks

Wie in Kapitel 1 beschrieben, soll der mit dieser Arbeit vorgestellte Compiler vollautomatisch sequentielle Programme auf parallel arbeitende Zellularautomaten abbilden. Zur Entwicklung paralleler Programme reduzieren sich die Aufgaben für Programmierer durch den Compiler auf die Planung und Erstellung von Algorithmen für Single-Core CPU-Systeme. Der Softwareentwickler entwirft den Algorithmus, schreibt das Programm in sequentieller Abfolge und gibt es an den Compiler weiter. Anders als bei den in Kapitel 2 beschriebenen Verfahren sind keine zusätzlichen Codewörter zur Parallelisierung oder Synchronisation notwendig. Darüber hinaus werden durch den Compiler keine expliziten parallelen Berechnungen in die Algorithmen eingefügt. Die parallele Verarbeitung entsteht als Nebenprodukt der Abbildung auf die Zellularautomaten. Das entspricht den in [62] beschriebenen Ideen für zukünftige Eingebettete Systeme. Nach Vorschlag der Autoren sollten sich Eingebettete Systeme in ihrer Charakteristik wie lebendige Organismen und Populationen verhalten; parallel arbeiten und sich selbstständig synchronisieren.

Zur Codierung der Algorithmen wurde ANSI-C als Programmiersprache gewählt. Der Grund dafür ist, dass viele zeitkritische Programme in dieser Sprache programmiert werden oder bereits existieren [70], deren Performance von einer automatischen Parallelisierung profitieren kann.

Das System folgt dem in [79] (siehe Kapitel 2.2.3) erläuterten Basis-Ansatz zur Bildung von Zellularautomaten aus sequentiellen Programmen. Das hier beschriebene Framework hat als Ziel jedoch nicht die Simulation von Zellular-

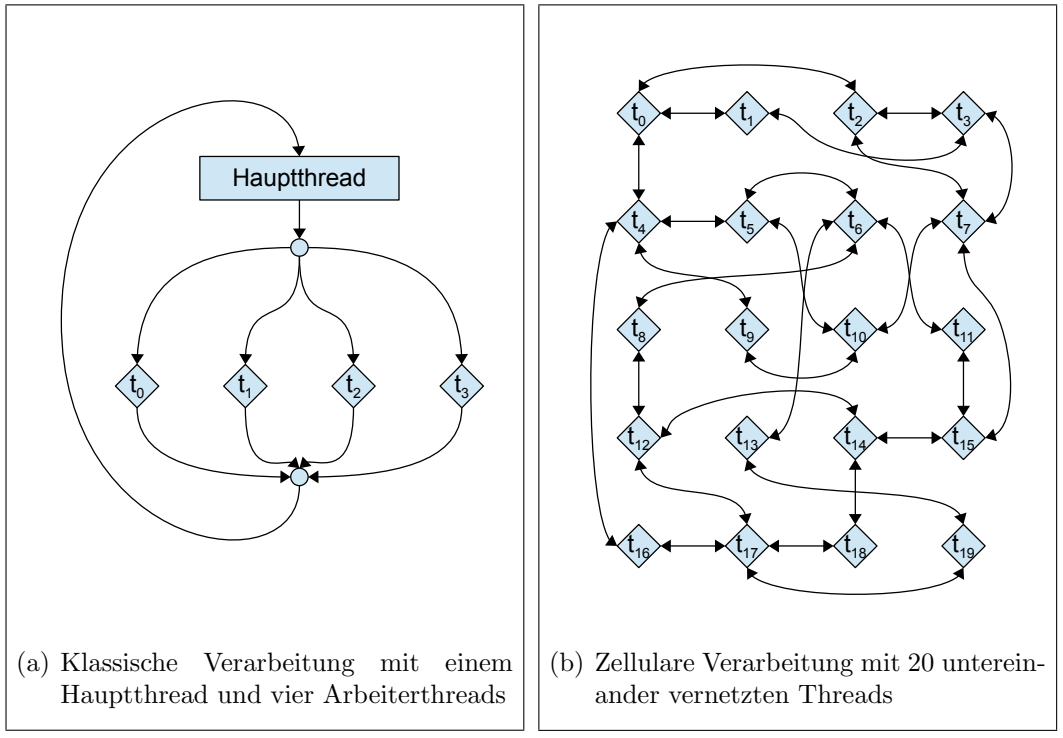
automaten. Vielmehr soll es dazu dienen, automatische Parallelität und damit die Nutzung vorhandener Ressourcen auf unterschiedlichen Hardwarearchitekturen zu ermöglichen.

Die Zellularautomaten werden bei dem vorgestellten Verfahren als Übergangssprache zwischen sequentiellem Programmcode und paralleler Software/Hardware genutzt. Aufgabe des Compilers ist es, Modelle von Zellularautomaten zu erzeugen, die bei gleichen Eingaben identische Ergebnisse wie das zugehörige C-Programm liefern. Zur Erreichung des Ziels wird vom Compiler zuerst der Programmfluss analysiert, um Datenabhängigkeiten zu ermitteln. Anschließend teilt der Compiler das Programm in einzelne Aufgaben und Verknüpfungen zum Datenaustausch ein.

Während bei klassischen Systemen Threads mehrere Befehle und somit vergleichsweise große Aufgaben bearbeiten, werden beim vorgestellten Zellularautomaten lediglich kleine Aufgaben, beispielsweise die Addition zweier Zahlen, an einen Thread übergeben. Die Threads informieren sich untereinander, wenn sie ihre Berechnungen abgeschlossen haben bzw. ab welchem Zeitpunkt ein anderer Thread die Daten übernehmen kann. Das so entstehende System unterscheidet sich im Grad der Granularität von dem in Kapitel 2.1.2 vorgestellten *SUIF Compiler* und den darauf aufbauenden Verfahren, beispielsweise dem *PEGASUS Projekt* oder dem in Kapitel 2.1.4 vorgestellten *CASH Compiler*. Während die genannten Verfahren aus mehreren Quellcodezeilen bestehende Blöcke innerhalb der Threads verarbeiten, arbeitet das in der vorliegenden Arbeit beschriebene System deutlich feingranularer.

Metaphorisch lassen sich klassische Multi-Thread Systeme mit Firmen vergleichen, die einen Vorgesetzten (Hauptprogramm) und nur wenige, aber sehr leistungsfähige, universell einsetzbare Mitarbeiter (Threads) haben. Der Vorgesetzte vergibt Teilaufgaben eines Problems an seine Mitarbeiter. Wenn diese ihre Aufgaben erfüllt haben, sammelt und verarbeitet der Vorgesetzte die Ergebnisse und vergibt anschließend neue Aufgaben an die Mitarbeiter. Sind einzelne Mitarbeiter dabei schneller als andere, führt das zwangsläufig zu Wartezeiten. Ihre hohe Leistungsfähigkeit ist in diesem Moment ungenutzt.

Das zellulare Bearbeitungsmodell folgt einem anderen Prinzip. Es gibt bei Zellularautomaten keinen Vorgesetzten. Die Mitarbeiter haben eine begrenzte Leistungsfähigkeit und sind nicht universell einsetzbar. Allerdings sind sie in deutlich höherer Anzahl vorhanden als im klassischen System. Jeder Mitarbei-



(a) Klassische Verarbeitung mit einem Hauptthread und vier Arbeiterthreads

(b) Zellulare Verarbeitung mit 20 untereinander vernetzten Threads

Abbildung 3.1: Vergleich zwischen klassischer und zellularer Verarbeitung eines Problems

ter kennt von Anfang an seine Funktion und weiß, von welchen Ergebnissen seine spezielle Aufgabe abhängt. Er meldet den dafür zuständigen Arbeitern, dass er die Ergebnisse benötigt und sobald sie vorhanden sind, erledigt er seine Aufgabe. Auch in diesem System müssen Einzelne zwischenzeitlich warten. Aufgrund ihrer niedrigeren Leistungsfähigkeit ist der Leistungsverlust für das Gesamtsystem jedoch geringer als beim klassischen Ansatz. Zusätzlich werden die Wartezustände durch die Vielzahl an Arbeitern und durch die Selbstsynchronisierung der Arbeiter untereinander, unabhängig von einem Vorgesetzten, minimiert.

In Abbildung 3.1 sind die Unterschiede zwischen klassischer und zellularer Verarbeitung eines Problems anhand von jeweils einem Beispiel dargestellt.

Ist die Erstellung des Modells des Zellularautomaten aus dem zugehörigen C-Programm abgeschlossen, kann es anschließend für unterschiedliche Zielarchitekturen übersetzt werden. In der vorliegenden Arbeit werden drei unterschiedliche Architekturen miteinander verglichen; zwei softwarebasierte Syste-

me, davon eines auf der CPU und eines auf der GPU, sowie ein hardwarebasiertes System auf einem FPGA.

3.2 Der C-Parser - Erzeugung von abstrakten Syntaxbäumen

Der erste wichtige Schritt zur Verarbeitung des vom Programmierer geschriebenen Algorithmus ist das Umwandeln des Codes in ein einfach weiterzuverarbeitendes Format. Diesen Arbeitsschritt übernimmt der *Parser*. Er liest die geschriebenen Worte, Zahlen sowie Symbole und interpretiert sie als Befehle, Daten und Variablen anhand von festgelegten Zeichenfolgen.

Zur Einhaltung und effizienten Speicherung der Programmstruktur werden die vom Parser ermittelten Programmteile in einer Baumstruktur abgelegt, die als *abstrakter Syntaxbaum* [73] bezeichnet wird. „Abstrakt“ steht dabei für die implizite Beschreibung von Abhängigkeiten der einzelnen Codeteile über die Verbindungen des Baumes. Beispielsweise wird die explizite Klammerung sowie Priorität von mathematischen Operationen im Quellcode durch Eltern/Kind-Beziehungen der Knoten im abstrakten Syntaxbaum ausgedrückt.

3.2.1 Die Benutzeroberfläche des Parsers

Der für die vorliegende Arbeit entwickelte C-Parser ist, wie der gesamte Rest des Frameworks, in C++ geschrieben und benutzt *Qt-Bibliotheken* [89] zur Darstellung der Benutzeroberfläche und zur Textverarbeitung.

In Abbildung 3.2 ist die Oberfläche des Programms und die Repräsentation eines Beispielcodes als Baumstruktur dargestellt. Das Benutzerinterface ist in drei *Widgets* unterteilt. Widget (1) zeigt die aktuelle Datei als Quellcode an. Neben der Darstellung des algorithmischen Ablaufs als abstrakter Syntaxbaum sind in Widget (2) die einzelnen Dateien als Kinderknoten des Projektes enthalten. Mit Klicken auf den jeweiligen Dateiknoten können Benutzer zwischen einzelnen Dateien wechseln. Durch Auswahl eines Unterknotens wird die entsprechende Stelle der aktuellen Datei in Widget (1) markiert. Das erleichtert die Navigation innerhalb des Projektes für den Benutzer und es können schnell Änderungen oder Korrekturen am Code vorgenommen werden.

Die erste Spalte der Baumansicht in Widget (2) enthält die Namen der Kno-

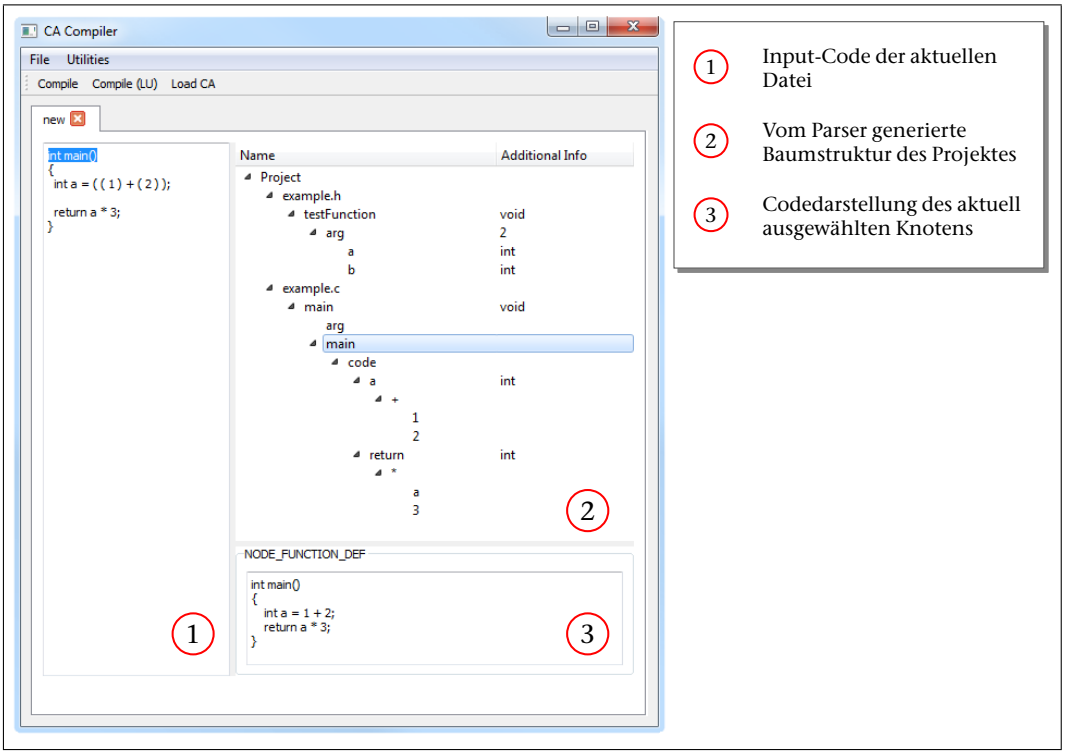


Abbildung 3.2: *C-Parser* - Codebeispiel und generierte Baumstruktur

ten. In der zweiten Spalte können bei bestimmten Knoten Zusatzinformationen enthalten sein. Dazu gehört unter anderem die Anzahl der Argumente einer Funktion sowie der Datentyp von Variablen und der Rückgabewert von Funktionen.

Widget (3), auf der unteren rechten Seite der Oberfläche, dient der detaillierten Anzeige des ausgewählten Knotens. Zusätzlich zur genauen Typenbezeichnung wird der durch den Knoten und die zugehörigen Unterknoten repräsentierte C-Code dargestellt. Dieser wird anders als in der Input-Darstellung aus der Baumstruktur generiert und ist auf die, für die Weiterverarbeitung notwendigen, relevanten Informationen reduziert. Abbildung 3.2 lässt das anhand der irrelevanten Klammern im Input-Code, die in der generierten Variante fehlen, erkennen. Kommentare des Programmierers sowie durch bedingte Präprozessoranweisungen (`#if`, `#ifdef`, `#ifndef`) ausgeschlossene Programmteile sind in dieser Ansicht ebenfalls nicht enthalten, da sie bei der Erzeugung des abstrakten Syntaxbaumes nicht übersetzt werden. Der Parser übernimmt somit zusätzlich die Aufgabe des klassischen C-Präprozessors, der für reine

Textveränderungen des Quellcodes, gesteuert durch Präprozessor-Direktiven, zuständig ist.

3.2.2 Abbildung von Formeln und Berechnungen in abstrakte Syntaxbäume

Zur korrekten Abbildung der Reihenfolgen von Operationen und Berechnungen werden diese von der im Quellcode verwendeten *Infixnotation* in eine *Präfixnotation* überführt. Während bei der Infixnotation die Operanden links und rechts des Operators stehen, wird bei der Präfixnotation zuerst der Operator und anschließend der linke, gefolgt vom rechten Operanden, geschrieben. In der Präfixnotation sind keine Klammern zur Kennzeichnung der Reihenfolgen von Operationen notwendig. Die korrekte Abarbeitungsfolge ergibt sich implizit durch die Anordnung der Operationen. Die Präfixnotation kann durch ihre spezielle Beschreibung von Beziehungen direkt in eine Baumstruktur gebracht werden und eignet sich daher zur Bildung abstrakter Syntaxbäume.

Das implementierte Verfahren, zur Umwandlung von Infix- in Präfixnotation, basiert auf dem sogenannten *Shunting-yard-Algorithmus* [26]. Der Algorithmus wurde ursprünglich von *Edsger Wybe Dijkstra* zur Umwandlung von Infix- in Postfixnotation entwickelt und 1961 vorgestellt. Wendet man ihn in leicht abgeänderter Form (Vertauschung des Verhaltens beim Lesen von öffnenden und schließenden Klammmern) auf die umgekehrte Elementreihenfolge der Eingabe an, so erhält man anstatt der Postfix- die Präfixnotation.

Das Verfahren benutzt einen Stack für die Operatoren und eine Ausgabeliste, die, wie der Operatorstack, nach dem *Last-In-First-Out (LIFO)-Prinzip* arbeitet. Das heißt, das Element, welches zuletzt geschrieben wurde, wird als erstes ausgelesen.

Umwandlung von Infix- in Präfixnotation (algorithmischer Ablauf):

1. Invertiere die Eingabe
2. Lese ein Element
   - handelt es sich um eine Zahl, einen Funktionsaufruf oder eine Variable:
     - schreibe das Element in die Ausgabe

- handelt es sich um eine öffnende Klammer:
  - entnehme so lange Operatoren aus dem Operatorstack und schreibe diese in die Ausgabe, bis der aktuelle Operator eine schließende Klammer ist
  - verwerfe beide Klammern
- handelt es sich um einen Operator:
  - entnehme so lange Operatoren aus dem Operatorstack und schreibe diese in die Ausgabe, bis der Operator auf dem Stack eine niedrigere Priorität hat als der neue Operator
  - schreibe den Operator auf den Operatorstack

3. Sind noch Elemente in der Eingabe vorhanden, gehe wieder zu Punkt 2
4. Schreibe die noch im Operatorstack enthaltenen Operatoren in die Ausgabe

Ist die Umwandlung in Präfixnotation abgeschlossen, kann daraus die gewünschte Baumstruktur generiert werden. Dazu wird im ersten Schritt an der aktuellen Stelle des abstrakten Syntaxbaumes (Position der Formel) das erste Element der Präfixnotation eingefügt und als momentaner Elternknoten markiert. Anschließend werden alle Elemente nacheinander bearbeitet und als Kinderknoten eingefügt. Ist ein Elternknoten vollständig belegt (beispielsweise eine Binäroperation mit zwei Kinderknoten), wird der Baum nach „oben“ durchlaufen, bis ein unvollständig belegter Elternknoten gefunden wird. Dort wird das Element eingefügt und wiederum als aktueller Elternknoten markiert.

Ein Beispiel zur Umwandlung einer Formel und Zuweisung an eine Variable, von Infix- in Präfixnotation, zeigt Tabelle 3.1. Die zugehörige generierte Baumstruktur ist in Abbildung 3.3 dargestellt.

3.3 Der Präcompiler - Aufbereitung und Optimierung der eingelesenen Algorithmen

Nachdem der Parser alle algorithmisch relevanten Informationen aus dem Quellcode extrahiert und in einen abstrakten Syntaxbaum umgewandelt hat, folgt mit dem Präcompiler der nächste Schritt des Kompilierungsprozesses.

Umzuwandelnde Formel:	$a = (\ 31 + 10 \ ) \cdot (\ 19 - 1 \ )$	
Invertierte Reihenfolge:	$) \ 1 - 19 \ (\ \cdot \ ) \ 10 + 31 \ (= a$	
Eingelesenes Element	Operatorstack	Ausgabe (Präfixnotation)
)		
1	)	
−	)	1
19	−)	1
(	−)	19 1
·		− 19 1
)	·	− 19 1
10	) ·	− 19 1
+	) ·	10 − 19 1
31	+) ·	10 − 19 1
(	+) ·	31 10 − 19 1
=	·	+ 31 10 − 19 1
a	=	· + 31 10 − 19 1
	=	a · + 31 10 − 19 1
		= a · + 31 10 − 19 1

Tabelle 3.1: Umwandlung einer Formel von Infix- in Präfixnotation

Infixnotation:
$a = (31 + 10) \cdot (19 - 1)$

Präfixnotation:
$= \ a \cdot + \ 31 \ 10 - 19 \ 1$

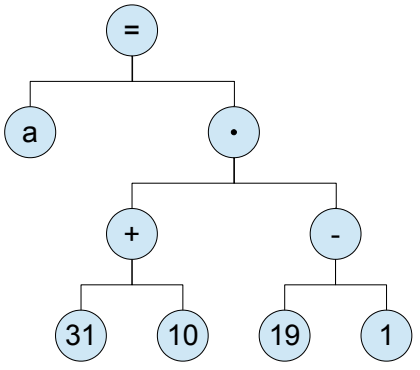


Abbildung 3.3: Aus Präfixnotation generierte Baumstruktur

Im Gegensatz zum klassischen Präprozessor, der bei dem beschriebenen Framework bereits im Parser enthalten ist (siehe Kapitel 3.2.1), arbeitet der in diesem Kapitel vorgestellte Präcompiler auf Basis der abstrakten

Durchgeführte Transformation	Beispiel	Ausgabe

Auflösung von Mehrfachzuweisungen	a = b = c = 5;	c = 5;
		b = c;
		a = b;

Ersetzung von Inkrement- und Dekrementoperationen	a++;	a = a + 1;
Auflösung zusammengesetzter Zuweisungen	a += 5;	a = a + 5;

Tabelle 3.2: Ersetzung von Operationen

Syntaxbäume, anstatt Textveränderungen am Quellcode vorzunehmen.

Die Aufgabe des Präcompilers besteht in der Aufbereitung und Vorverarbeitung der Algorithmen, um die anschließende Datenabhängigkeitsanalyse und Generierung des Zellularautomaten zu vereinfachen.

3.3.1 Ersetzung von Operationen

Eine erste Maßnahme, die zur Reduktion der im finalen Zellularautomaten notwendigen Zelltypen beiträgt, ist die Auflösung und Ersetzung von Operationen. Die durchgeführten Transformationen betreffen dabei ausschließlich für den Programmierer vereinfachende/abkürzende Schreibweisen bei der Zuweisung oder Veränderung von Variablen. Der Präcompiler löst die Abkürzungen auf und schreibt anschließend die neuen Befehle als Knoten in den abstrakten Syntaxbaum. Tabelle 3.2 gibt einen Überblick über die vom Präcompiler durchgeführten Aktionen.

3.3.2 Loop-Unrolling und andere Schleifenumformungen

In einem nächsten Arbeitsschritt des Präcompilers werden `for`-Schleifen mittels *Loop-Unrolling* aufgelöst - soweit das möglich und vom Benutzer gewünscht ist. Beim Loop-Unrolling werden die Schleifenvariablen mit den Werten der einzelnen Iterationen in den Schleifenkörper eingesetzt und diese so oft hintereinander kopiert, bis das Abbruchkriterium der Schleife erfüllt ist. Das dient vor allem der Steigerung des Parallelitätsgrades der finalen Anwendung. Anstatt die Iterationen der Schleife einzeln nacheinander zu durchlaufen, hat der generierte Zellularautomat durch das Verfahren die Möglichkeit, alle Iterationen gleichzeitig auszuführen, wenn die Datenabhängigkeiten es erlauben.

```
for ( i = 0; i < 10; i++ )
{
  doSomething( i );

}
```

```
i = 0;
while ( i < 10 )
{
  doSomething( i );
  i++;
}
```

Abbildung 3.4: Umwandlung einer `for`-Schleife in eine `while`-Schleife

Neben dem vollständigen Auflösen einer `for`-Schleife kann diese auch partiell, bis zu einer bestimmten Iteration, aufgelöst werden. Dadurch wird verhindert, dass der generierte Zellularautomat unverhältnismäßig groß wird. Die Kontrolle darüber hat der Benutzer; er wird vom System vor der Kompilierung gefragt, ob und in welchem Umfang Loop-Unrolling eingesetzt werden soll.

Zur Vermeidung einer unnötig hohen Anzahl an verschiedenen Elementen werden nicht aufgelöste `for`-Schleifen anschließend vom Präcompiler in `while`-Schleifen umgewandelt. Dazu wird die Initialisierung der Schleifenvariablen vor die Schleife gesetzt und die nach jeder Iteration auszuführenden Befehle werden an das Ende des Schleifenkörpers verschoben. Ein Beispiel für die Umwandlung einer `for`-Schleife in eine `while`-Schleife enthält Abbildung 3.4.

3.3.3 Optimierungen

Sind die für die Weiterverarbeitung notwendigen Maßnahmen abgeschlossen, folgt ein abschließender Optimierungsschritt des Präcompilers.

Eine der durchgeführten Optimierungen besteht in der Löschung nicht erreichbarer bzw. redundanter Teile des Algorithmus. Beim so genannten *Dead-Code-Elimination* [68] werden beispielsweise Variablen, denen zwar Werte zugewiesen, die jedoch nie ausgelesen werden, entfernt. Weitere entfernbare Elemente stellen unter anderem Befehle, die nach einer `return`-Anweisung stehen, sowie Codeblöcke innerhalb unerreichbarer Verzweigungen dar. Die durchgeführten *Dead-Code-Elimination*-Techniken im, für diese Arbeit entwickelten, Präcompiler beschränken sich auf die Löschung ungenutzter (nicht aufgerufener) Funktionen. Eine Erweiterung des Präcompilers um die genannten Verfahren ist für Weiterentwicklungen möglich, liegt jedoch außerhalb des Rahmens der vorliegenden Arbeit.

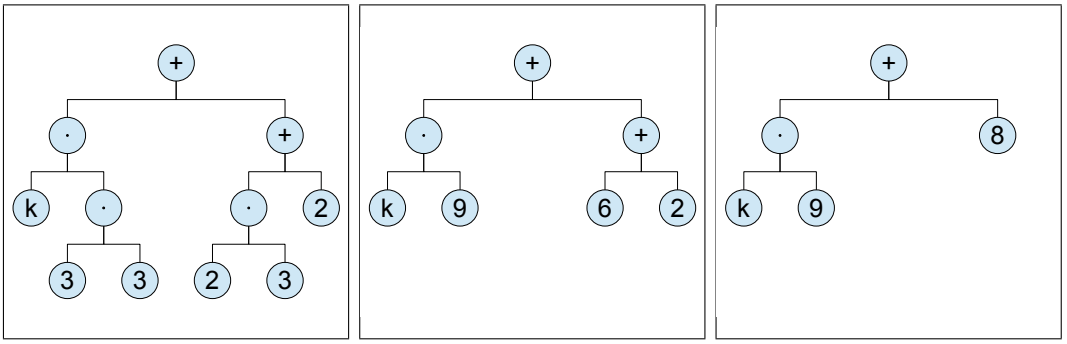


Abbildung 3.5: Rekursive Auswertung und Ersetzung einer Formel im abstrakten Syntaxbaum

Die Ersetzung von bereits zum Kompilierungszeitpunkt auswertbaren Berechnungen ist eine andere im Präcompiler implementierte Optimierungsmethode. Dabei werden Formeln vom Präcompiler auf ihre Auswertbarkeit untersucht. Solche Formeln können einerseits vom Programmierer, auf Grund besserer Lesbarkeit des Quellcodes, verwendet worden sein. Andererseits können sie durch die vorangegangenen Schritte des Präcompilers entstanden sein.

Ein Beispiel für einen solchen Fall stellen die auf eine Dimension reduzierten Arrayindizes dar (siehe Kapitel 4.1.1). Handelt es sich bei den ursprünglichen Indizes um konstante Werte, kann der neue Index durch Bildung der einzelnen Produkte und Summen bereits während der Kompilierung berechnet werden.

Implementiert wurde das Verfahren durch eine statische Methode, die einen Knoten als Eingabe erhält und dessen Wert rekursiv über seine Kinderknoten berechnet. Ist einer der Kinderknoten nicht auswertbar, gibt die Methode einen Fehlerwert zurück und der Knoten kann nicht durch einen konstanten Wert ersetzt werden.

Abbildung 3.5 veranschaulicht das Verfahren an einem Beispiel.

3.4 Datenabhängigkeitsanalyse

Im Anschluss an die durch den Präcompiler durchgeführten Operationen folgt in einem nächsten Kompilierungsschritt die Datenabhängigkeitsanalyse. Mit ihr sollen voneinander abhängige und unabhängige Teile des Programms identifiziert und entsprechend markiert werden. Das ist notwendig, um bei der finalen parallelen Ausführung des Zellularautomaten den vom Programmierer codierten Algorithmus korrekt ablaufen zu lassen.

```
1 int func( int a )
2 {
3
4
5   if ( a < 5 )
6     a = a + 1;
7
8   return a;
9 }
```

(a) Originaler C-Code

```
1 int func( int a_0 )
2 {
3   int a_1;
4
5   if( a < 5 )
6     a_1 = a + 1;
7
8   return a;
9 }
```

(b) Einfügen neuer Variablen für jede Zuweisung bzw. jeden Schreibzugriff

```
1 int func( int a_0 )
2 {
3   int a_1;
4
5   if( a_0 < 5 )
6     a_1 = a_0 + 1;
7
8   return a_?; // a_0 oder a_1?
9 }
```

(c) Ersetzung der Lesezugriffe

```
1 int func( int a_0 )
2 {
3   int a_1;
4
5   if( phi_0( a_0 ) < 5 )
6     a_1 = phi_1( a_0 ) + 1;
7
8   return phi_2( a_0, a_1 );
9 }
```

(d) Einführung von ϕ-Funktionen

Abbildung 3.6: Umwandlung von C-Code in SSA-Darstellung

3.4.1 Static-Single-Assignment-Darstellung (SSA)

Die SSA-Darstellung ist, wie in Kapitel 2.1.2 beschrieben, eine gebräuchliche Zwischensprache für Compileranalysen und Programmoptimierungen. Ihre Besonderheit liegt darin, dass Variablen in dieser Darstellung exakt einmal ein definierter Wert zugewiesen wird. In einem regulär von einem Programmierer erstellten Algorithmus ist das jedoch selten der Fall. Daher muss der Algorithmus verändert werden, um den SSA-Anforderungen zu entsprechen.

Abbildung 3.6 veranschaulicht in den Schritten (a) bis (d) die mehrstufige Umwandlung eines C-Codes (Abbildung 3.6(a)) in die zugehörige SSA-Darstellung an einem Beispiel.

Im ersten Transformationsschritt (Abbildung 3.6(b)) werden alle Zuweisungen an eine Variable identifiziert und für jede dieser Zuweisungen eine neue Variable angelegt. Dadurch ist die Bedingung, dass jeder Variable exakt einmal ein definierter Wert zugewiesen werden darf, erfüllt.

Der zweite Teilschritt des Verfahrens (Abbildung 3.6(c)) beseitigt die un-

gültig gewordenen Lesezugriffe auf die ursprüngliche Variable und ersetzt sie durch korrekte Zugriffe auf die neu erstellten Variablen. Problematisch ist das nach Verzweigungen im Algorithmus, an denen unterschiedliche Variablen zur Laufzeit gültig sein können. Ein solcher Fall tritt im Beispiel in Codezeile 8 auf. An dieser Stelle kommen sowohl `a_0` als auch `a_1` als einzusetzende Variablen in Frage.

Eine Lösung für das Problem stellt die Einführung so genannter ϕ-Funktionen dar (Abbildung 3.6(d)). Sie regeln den Zugriff auf die potentiell an den jeweiligen Codestellen möglichen Variablen-Instanzen einer ursprünglichen Variable, überprüfen die Instanzen auf ihre Gültigkeit und geben die korrekten Werte zurück.

Sind, wie im Beispiel für `a_0` < 5, mehrere Werte gültig, das heißt `a_0` und `a_1` sind mit definierten Werten beschrieben, entscheiden die ϕ-Funktionen anhand von *Use-Conditions* welcher Wert zurückgegeben wird. Diese Bedingungen werden während der Datenabhängigkeitsanalyse auf Basis des vom Programmierer geschriebenen Algorithmus ermittelt und in den jeweiligen ϕ-Funktionen gespeichert.

Eine Möglichkeit zur Bildung der Use-Conditions besteht in der direkten Einsetzung der im Algorithmus codierten Bedingungen für Verzweigungen. Das heißt, der Pfad im Code, der notwendig ist um die Variable zu erreichen, wird als Bedingung für ihre Verwendung eingesetzt. Im Beispiel in Abbildung 3.6 bedeutet das, dass `a_1` als Rückgabewert der ϕ-Funktion benutzt wird, wenn `a0` < 5 ist. Das ist korrekt, jedoch ohnehin Voraussetzung dafür, dass `a_1` einen gültigen Wert erhält. Variable `a_0` hat bei der Anwendung dieses Verfahrens keine vom Programmierer festgelegten Bedingungen und wird als Rückgabewert der ϕ-Funktion benutzt, sobald ein gültiger Wert vorliegt, was bereits beim Aufruf der Funktion gewährleistet ist. Zwar ist es möglich, dass durch die parallele Ausführung der Codeteile im Zellularautomaten beide Variablen beim Auswerten der ϕ-Funktion gültig sind und somit `a_1` gewählt wird, garantiert werden kann das durch dieses Verfahren allerdings nicht.

Ein anderes und im vorgestellten System implementiertes Verfahren, das zu einer korrekten Auswertung der Variablen durch Use-Conditions führt, nutzt die bedingten Pfade einer Variable zur Einschränkung des Zugriffs auf die anderen Variablen. Im Beispiel bedeutet das, anstatt den Zugriff auf `a_1` durch

eine Use-Condition mit `a_0` < 5 zu sichern, wird in der ϕ-Funktion eine Use-Condition für `a_0` mit `a_0` ≥ 5 gespeichert. Die Variable `a_1` braucht hier keine Use-Condition, da sie nur dann einen gültigen Wert erhält, wenn die algorithmische Bedingung `a_0` < 5 zutrifft. Durch dieses Verfahren wird sichergestellt, dass immer die korrekte Variable gewählt wird, unabhängig von der Ausführungsreihenfolge der einzelnen Codeteile.

Der Algorithmus als Pseudo-Code zur Ermittlung und Einsetzung der Use-Conditions ist in [79] beschrieben.

3.4.2 Schleifen in der SSA-Darstellung

Zur Erhöhung des Parallelitätsgrades des finalen Zellularautomaten sollten, wie in Kapitel 3.3.2 beschrieben, Schleifen möglichst durch Loop-Unrolling aufgelöst werden. Da das nur für Schleifen mit bereits zum Kompilierungszeitpunkt auswertbaren Schleifenvariablen möglich ist und der Benutzer sich zur Reduzierung der Programmgröße gegen Loop-Unrolling entscheiden kann, müssen Schleifen ebenfalls vom Compiler in die SSA-Darstellung und den Zellularautomaten übersetzt werden können.

Ein Vorteil der bereits vom Präcompiler durchgeführten Operationen besteht in der Beschränkung auf einen Schleifentyp: die `while`-Schleife. Zwar sind `do-while`-Schleifen ebenfalls möglich, sie werden jedoch nicht gesondert betrachtet. Der Zellularautomat verarbeitet sie als `while`-Schleifen, führt jedoch die erste Iteration ohne Auswertung und Überprüfung der Schleifenbedingungen aus.

Schleifen widersprechen der grundsätzlichen Struktur der SSA-Darstellung, da in jeder Schleifeniteration die selben Variablen beschrieben werden. Um zu verhindern, dass sich daraus ein Problem für die parallele Ausführung ergibt, müssen die Schleifenbedingungen und Schleifenkörper nach jeder Iteration explizit gelöscht werden. Abbildung 3.7(a) verdeutlicht, was ohne Zurücksetzen der Werte passieren kann.

In dem Beispiel kann bei paralleler Ausführung der Codezeilen 3 und 4 zur Berechnung von `y_1` der Wert von `x_1` aus der vorhergehenden Iteration benutzt werden, wenn er noch nicht durch den neuen Wert der aktuellen Iteration überschrieben wurde. Damit ist der Wert von `y_1` falsch und der Algorithmus würde nicht korrekt ablaufen.

```
1  while( ... )
2  {
3    x_1 = ... ;
4    y_1 = phi_0( x_1 ) * 5;
5
6    if ( ... )
7      x_2 = ... ;
8
9
10
11 }
```

(a) Code **ohne** Reload-Variablen

```
1  while( ... )
2  {
3    x_1 = ... ;
4    y_1 = phi_0( x_1 ) * 5;
5
6    if ( ... )
7      x_2 = ... ;
8
9    x_3 = phi_1( x_1, x_2 );
10   y_2 = phi_2( y_1 );
11 }
```

(b) Code **mit** Reload-Variablen

Abbildung 3.7: Einführung von *Reload-Variablen* zur Vermeidung iterationsübergreifender Fehler

Ein explizites Zurücksetzen der Schleife nach jeder Iteration verhindert das fehlerhafte Verhalten, führt jedoch zu neuen Problemen. Ist die Schleifeniteration von der vorhergehenden abhängig oder soll der Wert einer Variablen außerhalb der Schleife benutzt werden, so muss dieser gespeichert werden. Eine in [79] vorgestellte Möglichkeit zur Lösung des Problems stellen so genannte *Reload-Variablen* dar. Sie speichern exakt einen finalen Wert der alternativen Variablen am Ende jeder Iteration (Abbildung 3.7(b)). Reload-Variablen werden beim regulären Reset des umgebenden Schleifenkörpers nicht gelöscht. Sie werden nur dann ungültig, wenn eine äußere Schleife ihre inneren Schleifen zurücksetzt.

Die Einführung von Reload-Variablen löst dennoch nicht alle iterationsübergreifenden Probleme. Wird eine Variable zu Beginn des Schleifenkörpers gelesen, das heißt, es soll (bis auf die erste Iteration) der Wert aus der vorhergehenden Iteration übernommen werden, kann das ebenfalls zu fehlerhaften Werten führen. Das in Abbildung 3.8(a) dargestellte Beispiel veranschaulicht die potentiell fehlerbehaftete Situation.

```
1  x_0 = 0;
2
3  while( ... )
4  {
5
6    y_1 = phi_0( x_0, x_2 ) + 1;
7    x_1 = ... ;
8    x_2 = phi_1( x_1 );
9    y_2 = phi_2( y_1 );
10 }
```

(a) Code **ohne** Load-Variable

```
1  x_0 = 0;
2
3  while( ... )
4  {
5    x_1 = phi_0( x_0, x_3 ); // Load
6    y_1 = phi_1( x_1 ) + 1;
7    x_2 = ... ;
8    x_3 = phi_2( x_2 );
9    y_2 = phi_3( y_1 );
10 }
```

(b) Code **mit** Load-Variable

Abbildung 3.8: Einführung von *Load-Variablen* zur Vermeidung iterationsübergreifender Fehler

Das Problem entsteht beim Lesen der Variable `x_2` in Codezeile 6. Der Wert von `x_2` kann im Beispiel unabhängig von allen Berechnungen, die die Varianten der Variable `y` betreffen, berechnet werden. So wird es möglich, dass bereits zum Zeitpunkt der Auswertung von `y_1` ein neuer Wert von `x_2` der aktuellen Iteration gebildet ist. Dieser Wert wird durch die ϕ-Funktion `phi_0` zurückgegeben und führt zu einem fehlerhaften Verhalten des Algorithmus.

Eine Lösung für das Problem stellen die in Abbildung 3.8(b) eingefügten *Load-Variablen* dar. Sie übernehmen eine ähnliche Aufgabe wie die Reload-Variablen. Die Load-Variablen sorgen für einen eindeutigen Variableneingang zu Beginn jeder Schleifeniteration, werden im Gegensatz zu den Reload-Variablen jedoch beim Zurücksetzen des Schleifenkörpers in einen ungültigen Zustand versetzt.

Der Compiler kann auf das Einfügen von Load-Variablen verzichten, wenn ihre Funktion bereits von anderen Variablen übernommen wird. Dazu wird während der Datenabhängigkeitsanalyse überprüft, ob es einen Lesezugriff auf eine Variable gibt, bevor ein Schreibzugriff stattgefunden hat. Ist das der Fall, so muss eine Load-Variable eingefügt werden. Andernfalls kann darauf verzichtet werden, um die Anzahl an Variablen, Befehlen und Zuweisungen im kompilierten Zellularautomaten zu reduzieren.

Alle Lesezugriffe auf Variablen innerhalb der Schleife sind abhängig von den Load-Variablen, die wiederum auf die vor der Schleife gültigen Werte (für die erste Iteration) oder auf die Reload-Variablen zugreifen. Durch dieses Verfah-

```
1  int func()
2  {
3    int x_0, x_1, x_2;
4
5    x_0 = 0;
6    while( phi_0( x_0, x_2 ) < 10 )
7    {
8      x_1 = phi_1( x_0, x_2 ) + 1;
9      x_2 = phi_2( x_1 );
10   }
11
12   return phi_3( x_0, x_2 );
13 }
```

Abbildung 3.9: Auswertung von ϕ-Funktionen nach Schleifen

ren wird ein fehlerhaftes Vermischen der einzelne Iterationen verhindert und der korrekte Ablauf des Algorithmus gewährleistet.

Durch die Verwendung von Load- und Reload-Variablen ist es möglich, dass mehrere Variablen ohne oder mit identischen Use-Conditions gleichzeitig in einer ϕ-Funktion vorkommen. Ein Beispiel dafür stellt die Funktion `phi_3` in Abbildung 3.9 dar. Sowohl `x_0` als auch `x_2` haben die Use-Condition [`while == false`] und sind nach Abschluss der Schleife gleichzeitig gültig. Zur Unterdrückung eines der Argumente werden bei den in [79] eingesetzten ϕ-Funktionen zusätzlichen *Clear-Conditions* benutzt. Sie dienen dazu, Argumente der ϕ-Funktion unter bestimmten Bedingungen zu löschen, was notwendig ist, um die korrekte Wahl der Werte während der Laufzeit bzw. nach Abschluss von Schleifen zu gewährleisten. Im hier vorgestellten Verfahren wird darauf verzichtet und es wird die Variable gewählt, die später im Algorithmus vorkommt und dementsprechend weiter „hinten“ in der ϕ-Funktion steht.

Eine Optimierungsmaßnahme zur Reduktion der finalen Elemente erfolgt durch die Entfernung/Verlagerung von für alle Argumente einer ϕ-Funktion gültigen Bedingungen. Im Beispiel in Abbildung 3.9 muss in Codezeile 12 bei der Auswertung von `phi_3` zwischen `x_0` und `x_2` entschieden werden. Diese Auswertung darf erst dann stattfinden, wenn die Schleife vollständig ausgeführt wurde, um algorithmische Fehler zu verhindern. Zur Blockierung der Auswertung der Argumente von `phi_2` ist es notwendig, sowohl für `x_0` als

auch `x_2` eine Use-Condition [`while == false`] zu speichern. Je mehr Argumente die ϕ-Funktion hat, desto größer wird dieser zusätzliche Overhead. Zur Optimierung wird die ϕ-Funktion selbst mit der Bedingung als Voraussetzung versehen, anstatt die Bedingung für jedes Argument zu speichern. Die Auswertung der ϕ-Funktion wird somit erst gestartet, wenn ihre Bedingungen erfüllt sind.

3.5 Zellaufbau und Funktionsweise des Zellularautomaten

Im Anschluss an die Datenabhängigkeitsanalyse verwendet der Compiler den abstrakten Syntaxbaum in SSA-Darstellung, um daraus das Modell des Zellularautomaten zu generieren. Im Wesentlichen handelt es sich beim eingesetzten Modell um einen synchronen globalen Zellularautomaten (siehe Kapitel 2.2.2) mit statischer Nachbarschaft. Der Zellularautomat beinhaltet verschiedene Zelltypen, die sich in ihren Regeln zur Bildung ihrer neuen Zustände auf Basis der vorhergehenden unterscheiden.

3.5.1 Unterschiedliche Zelltypen und ihre Aufgaben

Abgeleitet ist die Funktionalität der Zellen aus den in der SSA-Darstellung vorkommenden Elementen. Jedem Element im abstrakten Syntaxbaum wird eine Zelle zugeordnet. Durch die Übernahme der Baumstruktur haben die Zellen im Allgemeinen drei Nachbarn: eine übergeordnete Elternzelle, von der die Zelle Anweisungen zur Ausführung erhält sowie zwei Kinderzellen, von deren Ausführung die Zelle selbst abhängig ist und an die sie Befehle weitergibt. Das bedeutet, dass die Mehrzahl der Zellen keine globalen Kommunikationsverbindungen zu anderen Zellen benötigen. Stattdessen kommunizieren sie lokal mit ihren direkten Nachbarn.

Neben Funktionen, die ihren Rückgabewert aus einer entfernten Zelle auslesen, müssen lediglich Zellen, die ϕ-Funktionen repräsentieren, auf entfernte Zellen, die ihre Argumente oder deren Bedingungen darstellen, zugreifen und ihre aktuellen Zustände und Werte auslesen können.

Zur Realisierung eines Zellularautomaten, der in der Lage ist, einfache C-Algorithmen (ohne Arrays, Pointer und rekursive Funktionen) darzustellen und auszuwerten, sind neun verschiedene Zelltypen ausreichend. Dazu gehören

neben den bereits erwähnten ϕ-Funktionen:

- *Weiterleitungszellen* [fork] - sie leiten einen Befehl an ihre Kinderzellen weiter und warten auf deren Antwort,
- *Funktionszellen* [func] - sie leiten die in die Funktionsargumente einzusetzenden Werte an die entsprechenden Zellen weiter und liefern nach Auswertung des Funktionsrumpfes einen Rückgabewert,
- *Verzweigungszellen* [if] - sie realisieren die, durch **if-then-else**-Anweisungen erzeugten, alternativen Ausführungspfade und steuern deren Ablauf,
- *binäre Operationszellen* [binOp] - sie setzen ihren eigenen Wert auf die mittels ihrer zugewiesenen Operation verknüpften Werte ihrer beiden Kinderzellen,
- *unäre Operationszellen* [unOp] - sie setzen ihren Wert auf den mittels ihrer zugewiesenen Operation veränderten Wert ihrer Kinderzelle,
- *Reload-Zellen* [reload] - sie dienen der expliziten Speicherung eines Variablenwertes, der iterationsübergreifend verwendet werden soll,
- *Wertezellen* [value] - sie liefern einen, zum Kompilierungszeitpunkt bekannten, konstanten Wert,
- *Schleifenzellen* [while] - sie repräsentieren `while`- und `do-while`-Schleifen und steuern deren mehrfache Ausführung.

Auf eine Umsetzung der expliziten Zuweisungen an Variablen, die im abstrakten Syntaxbaum existieren, wird aus Optimierungsgründen verzichtet. Ihre einzige Aufgabe, die Weiterleitung eines Wertes an eine Variable, kann durch eine geringfügige Umstrukturierung des abstrakten Syntaxbaumes, wie in Abbildung 3.10 dargestellt, erreicht werden.

Des Weiteren kann auf die Kompilierung der Variablen selbst, abgesehen von Reload-Variablen, verzichtet werden. Ihre Funktion, die Speicherung des in ihrer Kinderzelle vorhandenen Wertes, wird von der Kinderzelle selbst übernommen. Die Operations-, Werte- und Funktionszellen auf der rechten Seite der Zuweisung repräsentieren somit zusätzlich zu ihrer eigenen Funktion die

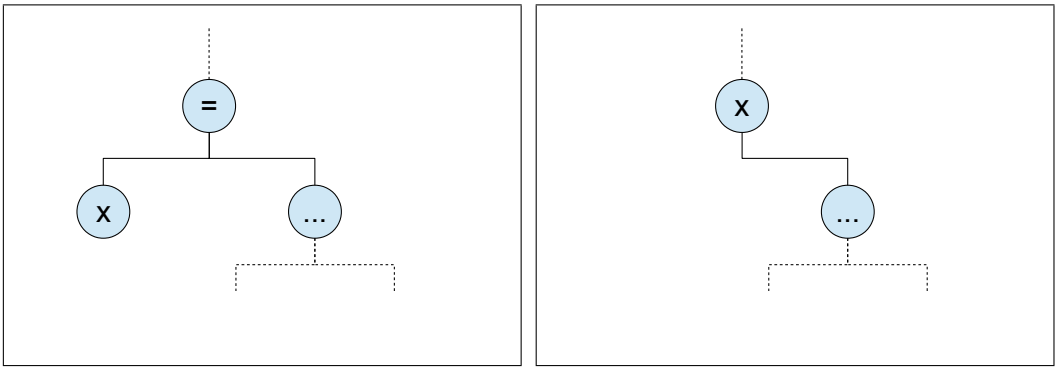


Abbildung 3.10: Entfernung von Zuweisungsknoten durch Umstrukturierung des abstrakten Syntaxbaumes

Variablenzelle. Alle ϕ-Funktionen, die Zugriff auf die Variable benötigen, werden vom Compiler entsprechend angepasst.

Reload-Variablen müssen explizit als Zellen erzeugt werden, da sie sich, wie in Kapitel 3.4.2 beschrieben, beim Zurücksetzen von Schleifen anders verhalten als andere Zelltypen. Eine Erweiterung aller in einer Gleichung möglichen Zelltypen auf den potenziellen Einsatz als Reload-Variable ist nicht umsetzbar, da das den Zweck ihrer Einführung (siehe Kapitel 3.4.2) aufheben würde. Ein zusätzlicher Speicherort für iterationsübergreifende Werte ist zwingend erforderlich und wird durch die Erzeugung von Reload-Zellen realisiert.

3.5.2 Zellkommunikation und Auswertung von Algorithmen im Modell des Zellularautomaten

Die Auswertung eines im Modell des Zellularautomaten codierten Algorithmus folgt immer dem gleichen Prinzip. Eine Zelle erhält den Befehl zur Berechnung ihres Ergebnisses. Ist das Ergebnis von ihren Kinderzellen abhängig, leitet sie den Befehl zur Auswertung an diese Zellen weiter. Haben die Kinderzellen ihre Berechnungen abgeschlossen, kann die Elternzelle ihr Ergebnis berechnen und zurückgeben. Es handelt sich somit um ein rekursives Berechnungsmodell.

Die Besonderheit des Verfahrens liegt darin, dass alle aktivierten Zellen parallel arbeiten und sich gegenseitig über Aufgaben und Ergebnisse informieren. Dadurch entsteht ein selbstsynchronisierendes paralleles System, das implizit durch die in den Zellen codierten Datenabhängigkeiten und Kontrollstrukturen gesteuert wird.

Zustand des Senders	Nachricht	Aktion des Empfängers
Computing	*Evaluate*	Zustandswechsel: *Idle* → *Computing*; Auswertung/Berechnung der Zellregel wird gestartet
EvalTrue bzw. *EvalFalse*	*Evaluated*	Kinderzelle hat ihre Funktion abgeschlossen, eigene Zellregel kann (in Abhängigkeit von den anderen Kinderzellen) ausgewertet werden
Clearing	*Clear*	Zustandswechsel: *EvalTrue*/*EvalFalse* → *Clearing*; Zurücksetzen der Zelle wird gestartet, Weiterleitung der Nachricht an alle Kinderzellen
ClearingWR	*ClearWR*	Zustandswechsel: *EvalTrue*/*EvalFalse* → *ClearingWR*; Zurücksetzen der Zelle inklusiver eventueller Reload-Variablen wird gestartet, Weiterleitung der Nachricht an alle Kinderzellen

Tabelle 3.3: Nachrichten zwischen Zellen und ihre Auswirkungen

Die Kommunikation der Zellen untereinander basiert auf Nachrichten, die einen Zustandswechsel bzw. eine Reaktion der Empfängerzellen auslösen. Tabelle 3.3 gibt eine Übersicht über die verschiedenen Nachrichtentypen und die Aktionen, die sie beim Empfänger initiieren.
Die möglichen Zustände einer Zelle, in der zu erwartenden Reihenfolge, sind im Allgemeinen:

1. *Idle* - Warten auf Nachrichten,
2. *Computing* - Auswertung der eigenen Zellregel, eventuell Warten auf Ergebnisse der Kinderzellen,
3. *EvalTrue*/*EvalFalse* - Auswertung der Zellregel ist beendet und mit `true` oder `false` (nur bei Verzweigungs- oder Schleifenzellen) abgeschlossen,
4. *Clearing* - Zurücksetzen der Zelle,
5. *ClearingWR* - Zurücksetzen der Zelle inklusive eventueller Reload-Variablen.

Im Gegensatz zu den rein auf gerichteten Datenflussgraphen basierenden Konzepten aus Kapitel 2.1.2 werden in dem vorgestellten zellularen Modell bidirek-

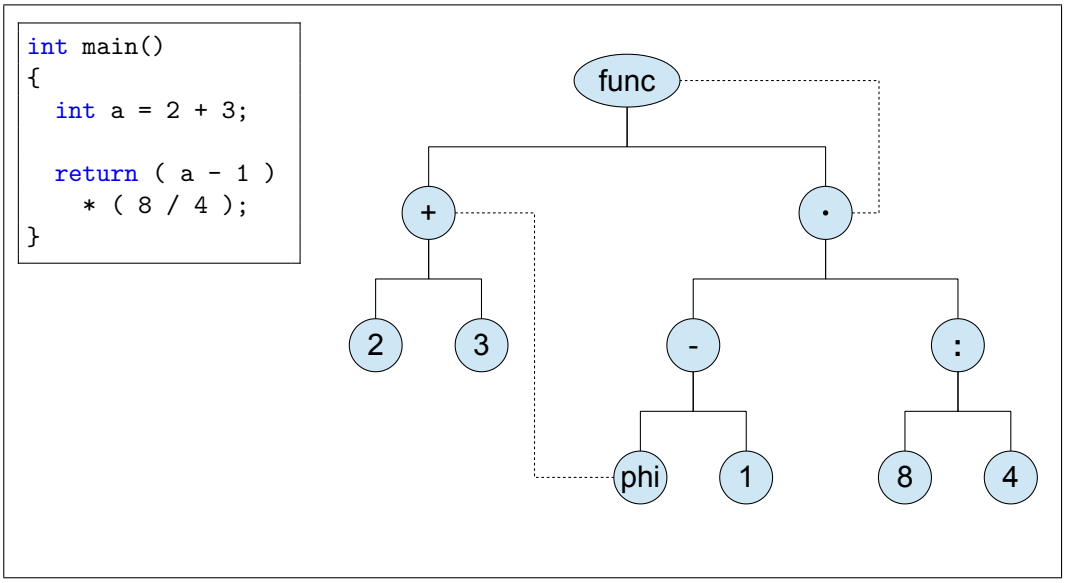


Abbildung 3.11: C-Code und daraus generiertes zellulares Modell

tionale Graphen verwendet. Alle Zellen melden, wenn sie ihre Berechnungen abgeschlossen haben. Das vermeidet zusätzliche Sammelknoten für Ergebnisse (Join) im Anschluss an alternative Verzweigungen.

Zur sicheren parallelen Ausführung wird im verwendeten Modell nur lesend von einer Zelle auf andere Zellen zugegriffen. Daher werden im Unterschied zu den in *PEGASUS* eingesetzten (siehe Kapitel 2.1.2) Multiplexor-Knoten, ϕ-Funktionen verwendet. In der entwickelten Variante haben sie zusätzlich den Vorteil der Vermeidung unnötiger Kopiervorgänge, da sie nur die Werte der Argumente laden, die aufgrund ihrer Use-Conditions tatsächlich notwendig sind. Das unterscheidet das Verfahren zu dem in [79], bei dem alle ausgewerteten Argumente, ähnlich zu der Benutzung von Multiplexor-Knoten, an die ϕ-Funktion versendet (kopiert) werden und anschließend anhand der Bedingungen innerhalb der ϕ-Funktion ein Argument ausgewählt wird.

Abbildung 3.11 zeigt einen Beispiel-Algorithmus und den aus dem Algorithmus generierten Zellularautomaten in Graphenform. Im Beispiel wurde zu Demonstrationszwecken auf den Einsatz der in Kapitel 3.3.3 vorgestellten Optimierungsverfahren des Präcompilers verzichtet. Daher sind Berechnungen enthalten, die normalerweise bereits vom Präcompiler ausgerechnet und durch das Ergebnis ersetzt wären.

Die folgenden Schritte zur Auswertung des Beispielmodells sind nummeriert und entsprechen den einzelnen Updates des Zellularautomaten, bis der Algorithmus vollständig abgearbeitet ist. Unterpunkte innerhalb eines Schrittes werden von den beteiligten Zellen parallel ausgeführt.

Algorithmischer Ablauf zur Auswertung des Beispielmodells:

0. alle Zellen befinden sich im *Idle*-Zustand und warten auf *Evaluate*-Nachrichten, die die Auswertung ihrer individuellen Zellregeln starten

1. wird der Zellularautomat gestartet, setzt die Funktionszelle [func] ihren Zustand auf *Computing* und sendet *Evaluate*-Nachrichten an ihre Kinderzellen

2. a) die Operatorzelle [+] setzt ihren Zustand auf *Computing* und sendet *Evaluate*-Nachrichten an ihre Kinderzellen
   b) die Operatorzelle [$\cdot$] setzt ihren Zustand auf *Computing* und sendet *Evaluate*-Nachrichten an ihre Kinderzellen

3. a) die Wertezelle [2] setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle
   b) die Wertezelle [3] setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle
   c) die Operatorzelle [$-$] setzt ihren Zustand auf *Computing* und sendet *Evaluate*-Nachrichten an ihre Kinderzellen
   d) die Operatorzelle [:] setzt ihren Zustand auf *Computing* und sendet *Evaluate*-Nachrichten an ihre Kinderzellen

4. a) die Operatorzelle [+] wertet die Ergebnisse ihrer Kinderzellen aus und bildet ihren eigenen Wert ($2+3=5$), setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle
   b) die ϕ-Zelle [phi] setzt ihren Zustand auf *Computing*
   c) die Wertezelle [1] setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle
   d) die Wertezelle [8] setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle

e) die Wertezelle [4] setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle

5. a) die ϕ-Zelle [phi] liest den Wert der Operatorzelle [+], setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle

   b) die Operatorzelle [:] wertet die Ergebnisse ihrer Kinderzellen aus und bildet ihren eigenen Wert ($8 : 4 = 2$), setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle

6. die Operatorzelle [$-$] wertet die Ergebnisse ihrer Kinderzellen aus und bildet ihren eigenen Wert ($5 - 1 = 4$), setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle

7. die Operatorzelle [$\cdot$] wertet die Ergebnisse ihrer Kinderzellen aus und bildet ihren eigenen Wert ($4 \cdot 2 = 8$), setzt ihren Zustand auf *EvalTrue* und sendet eine *Evaluated*-Nachricht an ihre Elternzelle

8. die Funktionszelle [func] liest den Wert der Operatorzelle [$\cdot$] und setzt ihren Zustand auf *EvalTrue*

Obwohl der originale C-Code rein sequentiell geschrieben ist, können die verarbeitenden Zellen in den Schritten 2 bis 5 parallel arbeiten und unabhängige Berechnungen gleichzeitig ausführen. Das betrifft beispielsweise binäre Operationen, bei denen beide Operanden gleichzeitig ausgewertet werden können. Fehlen Zwischenergebnisse für die Weiterverarbeitung, so warten die Zellen bis diese verfügbar sind. Ein solches Verhalten ist in regulärem C-Code nicht darstellbar. Darin besteht der Vorteil der Verarbeitung mittels selbstsynchronisierender Zellularautomaten.

3.6 Implementierung in Software

Im Anschluss an den theoretischen Aufbau des Zellularautomaten und die Beschreibung der Funktionsweise beschäftigt sich dieses Kapitel mit der Implementierung und Auswertung der Zellularautomaten in Software. Dabei werden zwei verschiedene Zielarchitekturen unterstützt; ein Multi-Core-System auf der CPU sowie ein Many-Core-System auf der GPU.

Unabhängig vom gewählten Zielsystem erfolgt die Ausführung und Ergebnisdarstellung mit dem selben Programm. Das Programm verwaltet die virtuellen Maschinen auf der CPU und GPU, die die Zellupdates durchführen und die Zellregeln auswerten. Anschließend liest es die Ergebnisse ein und stellt sie für den Benutzer dar.

3.6.1 Umwandlung der Baumstrukturen in zweidimensionale Zellularautomaten

In einem ersten Schritt zur effizienten Speicherung und Auswertung der Zellen werden sie, wie bei den in Kapitel 2.2 vorgestellten klassischen Zellularautomaten üblich, in ein zweidimensionales Gitter abgebildet.

Die Anzahl der Zellen pro Zeile bzw. Spalte des Gitters sollte möglichst ausgewogen sein. Das erleichtert die Ausführung des Zellularautomaten auf Systemen, bei denen die Anzahl der Zellen größer ist als die Anzahl der verfügbaren Recheneinheiten. Eine ausgewogene Verteilung auf Zeilen und Spalten vereinfacht die Verbindung von Zellen zu Blöcken, die anschließend von allen zur Verfügung stehenden Prozessoren parallel verarbeitet werden können. Damit wird eine gleichmäßige Auslastung und eine hohe Performance des Gesamtsystems erreicht.

Die folgende Formel wird zur Ermittlung der Zeilenbreite w aus der Gesamtzahl der Zellen n verwendet:

$$w = \left\lfloor \sqrt{n} + 0.5 \right\rfloor \ .$$

Die Höhe h des Zellularautomaten ergibt sich anschließend aus der Gesamtzahl n und der Zeilenbreite w mit:

$$h = \left\lceil \frac{n}{w} \right\rceil \ .$$

Die Formeln bewirken, dass bei steigender Anzahl an Zellen die Zellularautomaten zuerst in der Höhe und anschließend in der Breite expandieren. Es ist ebenfalls möglich die Formeln zu vertauschen. Dadurch würde der Zellularautomat zuerst in der Breite und anschließend in der Höhe wachsen.

Abbildung 3.12 zeigt die Ausdehnung des Gitters in Höhe und Breite für unterschiedlich große Zellularautomaten.

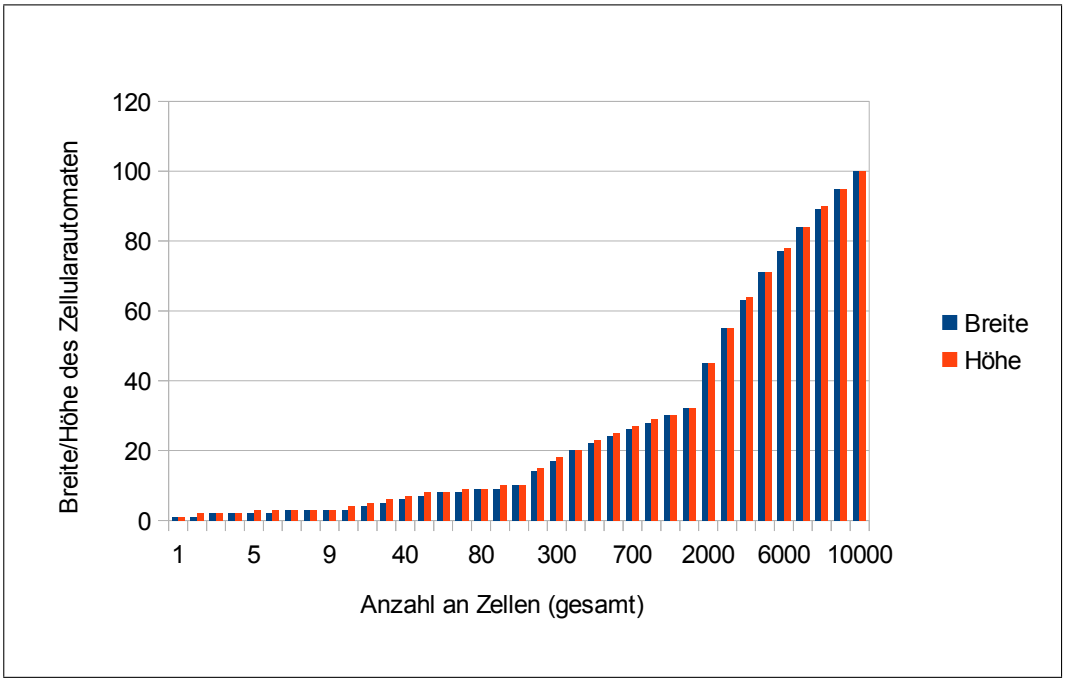


Abbildung 3.12: Ausdehnung von Zellularautomaten in Höhe und Breite bei unterschiedlicher Gesamtzahl an Zellen

Im Anschluss an die Bestimmung der Breite und Höhe des Zellularautomaten folgt die Abbildung der Zellen auf das erzeugte Gitter. Dazu wird die Baumstruktur mittels einer Tiefensuche traversiert und die untersuchten Knoten zeilenweise in das Gitter geschrieben. Das Verfahren entspricht dem in [94] beschriebenen, ohne zusätzliche Gewichte der Knoten des Baumes. Die Gitterstruktur des Zellularautomaten wird in der ausführbaren Software durch ein zweidimensionales Array realisiert.

Abbildung 3.13 zeigt die Umwandlung der Beispiel-Baumstruktur aus Kapitel 3.5.2 in einen zweidimensionalen Zellularautomaten.

3.6.2 Implementierung der Zellen und Kommunikationswege

Alle im Zellularautomaten möglichen Zelltypen leiten sich aus einer Basisklasse ab, die sowohl einen Zeiger auf den Wert der Zelle sowie Zeiger auf die jeweiligen Kinderzellen beinhaltet. Des Weiteren umfasst sie Methoden zum Setzen und Auslesen des Zellzustandes. Neben den Basisfunktionen und Variablen beinhaltet jeder daraus abgeleitete Zelltyp eine individuelle *Update*-Methode,

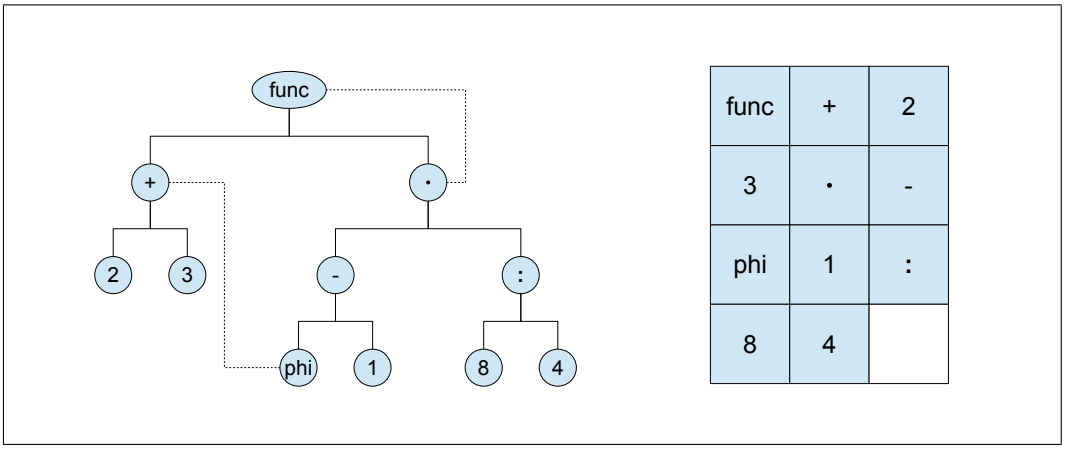


Abbildung 3.13: Umwandlung einer Baumstruktur in einen zweidimensionalen Zellularautomaten

die die spezifischen Zellregeln implementiert.

Im Unterschied zu dem in Kapitel 3.5 beschriebenen theoretischen Ansatz, findet bei der Auswertung der Zellen kein Nachrichtenaustausch im klassischen Sinn statt. Jede Elternzelle hat Zugriff auf einen individuellen Puffer ihrer Kinderelemente, der zur Übermittlung von Nachrichten bzw. zum Setzen des nächsten Zustandes der Kinderzellen genutzt wird.

Die Übermittlung von Nachrichten bzw. die Mitteilung des aktuellen Zustandes der Kinderzellen an ihre zugehörige Elternzelle passiert ebenfalls über einen Puffer, der von der jeweiligen Kinderzelle geschrieben und bei Bedarf von der Elternzelle überprüft und ausgelesen wird. Um möglichst wenig Speicherplatz für die Nachrichten zu verwenden, gibt es in der Implementierung keine Unterscheidung zwischen Nachrichten und Zuständen. Jede Zelle beinhaltet lediglich einen Eingabe- und einen Ausgabepuffer zum Lesen bzw. Schreiben des gewünschten bzw. erreichten Zustandes. Nach dem Schreiben in den Eingabe- bzw. Ausgabepuffer ist im Allgemeinen ein Update/Generationswechsel des Zellularautomaten notwendig, um den Zustand auszulesen. Das vermeidet den gleichzeitigen ungesicherten Zugriff verschiedener Threads auf das selbe Element. Eine Ausnahme bzw. Optimierung des Verfahrens ist in Kapitel 3.6.3 beschrieben.

Es werden insgesamt sechs verschiedene Zustände, analog zu den in Kapitel 3.5.2 beschriebenen Nachrichten und Zuständen, unterschieden und als jeweils

ein Byte für den Eingabe- und Ausgabepuffer in den Zellen gespeichert.

Die Zustände mit ihrem zugehörigen Wert sind:

- 0x00 $\widehat{=}$ *CLEARING*,
- 0x01 $\widehat{=}$ *CLEARING_WR*,
- 0x02 $\widehat{=}$ *UNDEFINED* (= *IDLE*),
- 0x03 $\widehat{=}$ *COMPUTING*,
- 0x04 $\widehat{=}$ *EVAL_TRUE*,
- 0x05 $\widehat{=}$ *EVAL_FALSE*.

Die Reihenfolge der verschiedenen Werte der Zustände ist relevant für eine optimierte Verarbeitung in den Zellen. Sie ermöglicht es, eine teilweise unnötige Unterscheidung der einzelnen Löschzustände und der *EVAL_TRUE/ EVAL_FALSE*-Zustände zu reduzieren. Ob sich eine Zelle in einem Löschzustand befindet, ist mittels `state < UNDEFINED` überprüfbar. Der Test, ob eine Zelle ihre Berechnungen abgeschlossen hat, kann mittels `state > COMPUTING` geschehen. Eine exakte Unterscheidung muss nur innerhalb spezieller Zelltypen stattfinden, zum Beispiel beim Zurücksetzen einer *Reload*-Zelle oder bei der Auswertung von Bedingungen in ϕ-Zellen.

Die Codierung der einzelnen Zustände kann auch mit weniger als einem Byte pro Puffer erfolgen, da zur Codierung der sechs verschiedenen Zustände lediglich drei Bits benötigt werden (Codierung mit drei Bits: acht verschiedene Werte möglich, da $2^3 = 8$). In C/C++, das zur Programmierung der Software eingesetzt wurde, existiert jedoch kein Standard-Datentyp, der weniger als acht Bits (ein Byte) umfasst. Die einzige Möglichkeit, eine solche Speicherplatzoptimierung in C/C++ durchzuführen, besteht in der Kombination des Eingabe- und Ausgabepuffers in einer Struktur (`struct`). Innerhalb der Struktur können Bitfelder mit einer vorgegebenen Anzahl an Bits definiert werden, so dass beide Puffer innerhalb eines Bytes gespeichert werden können. In der aktuellen Version des Compilers wurde darauf verzichtet. Die Kombination der Puffer in einer Struktur bietet daher Optimierungspotenzial für mögliche Weiterentwicklungen.

3.6.3 Globale und asynchrone Zellkommunikation

Im Gegensatz zu dem in Kapitel 3.5.2 beschriebenen Verfahren zum Zurücksetzen der Zellen durch Nachrichten, die rekursiv an alle Kinderzellen weitergeleitet werden, wird in der Softwareimplementierung ein optimiertes Verfahren zum Zurücksetzen der Zellen eingesetzt.

Das Löschen der Zellwerte kann, abgesehen vom Zurücksetzen des gesamten Zellularautomaten, lediglich von Zellen eingeleitet werden, die Schleifen repräsentieren. Das führt dazu, dass eine Zelle, die sich in der Hierarchie innerhalb eines Schleifenkörpers befindet, ausgehend von der zugehörigen *while*-Zelle den Befehl zum Zurücksetzen erhält. Bis diese Nachricht auf dem Weg durch die Hierarchie bei der Empfängerzelle angekommen ist, sind je nach Anzahl der Elternzellen, die durchlaufen werden müssen, mehrere Updates des Zellularautomaten nötig.

Zur Reduzierung des Overheads, der durch die Weiterleitung entsteht, wird innerhalb der Zellen ein zusätzlicher Zeiger verwendet, der mit ihrem Reset-Signal verbunden ist. Das Reset-Signal wird durch den Ausgabepuffer der nächsthöheren Schleifenzelle gesetzt. Ändert die Schleifenzelle ihren Ausgabewert auf *CLEARING*, so wird das in der nächsten Generation des Zellularautomaten von allen mit dem Reset-Signal verbundenen Zellen erkannt und sie können ihre Werte zurücksetzen.

Erhält eine innere Schleife ein Reset-Signal durch eine sie umgebende äußere Schleife, so muss sie ihren Ausgabezustand auf *CLEARING_WR* setzen, um alle in ihrem Schleifenkörper vorhandenen Zellen und *Reload*-Zellen korrekt zurückzusetzen.

Eine andere Optimierungsmaßnahme betrifft hauptsächlich Systeme mit weniger zur Verfügung stehenden Recheneinheiten/Prozessoren als Zellen im Zellularautomaten. Sie basiert darauf, dass jeder Prozessor mehrere Zellen bearbeiten muss und somit eine zeitliche Verzögerung innerhalb des Zellularautomaten existiert. Dieser Umstand ermöglicht es, eine teilweise asynchrone Kommunikation der Zellen zur Beschleunigung der Auswertung zu nutzen.

Erhält eine Zelle in ihrem Eingabepuffer den Zustand *COMPUTING* und wird aktiviert, kann der dafür eingesetzte Prozessor die Zelle auswerten. Ist sie von ihren Kinderzellen abhängig, schreibt sie ebenfalls den Zustand *COMPUTING* in die Eingabepuffer der Kinderzellen. Bei synchroner Kommunikation

der Zellen können die Kinderzellen diesen Zustand erst nach dem nächsten Update des Zellularautomaten auslesen. Es werden mehrere Updates notwendig, um eine Berechnung zu starten.

Bei einem vollständig parallel arbeitenden System, bei dem jeder Zelle eine individuelle Recheneinheit zugewiesen ist, hat das nur geringfügige Auswirkungen. Die Zellen überprüfen bei jedem Update erst ihren Eingabepuffer und führen anschließend ihre Berechnungen aus. Somit ist eine Überschneidung bei gleich schnell arbeitenden Prozessoren nahezu ausgeschlossen.

Verwaltet jeder Prozessor hingegen mehrere Zellen, so kann es passieren, dass eine Zelle, deren Berechnung von ihren Kinderzellen abhängig ist, bereits in die Eingabepuffer der Kinderzellen geschrieben hat, bevor diese im aktuellen Updateschritt des Zellularautomaten bearbeitet wurden. Das Warten auf das nächste Update des Zellularautomaten stellt in diesem Fall einen unnötigen Verlust von Rechenleistung dar. Um das zu vermeiden, werden die Eingabezustände direkt (ohne Zwischenpuffer) verwendet, unabhängig davon, aus welcher Zellgeneration sie stammen.

Hierzu wäre auch ein Verfahren, wie beim Zurücksetzen der Zellen, denkbar, bei dem eine übergeordnete Zelle einen Wert verändert und alle Zellen, die sich tiefer in der Hierarchie befinden, über einen Zeiger benachrichtigt werden. Im Gegensatz zu den, das Reset-Signal auslösenden, *while*-Zellen können diese Signale jedoch von unterschiedlichen Zelltypen ausgelöst werden. In der Softwareimplementierung wurde auf die Verwendung eines solchen Verfahrens verzichtet. Es könnte in zukünftige Weiterentwicklungen einfließen. Eine ähnliche Technik wird im aktuellen Framework allerdings für die Hardwareimplementierung eingesetzt (siehe Kapitel 3.7.2).

Die auf diese Weise realisierte asynchrone Kommunikation am Eingabepuffer der Zellen kann durch atomare Operationen beim Lesen und Schreiben umgesetzt werden, so dass die Datenkonsistenz der Zustände sichergestellt ist. Beim Schreiben des Ausgabepuffers wird dagegen auf asynchrones Schreiben und Lesen verzichtet. Neben dem Ausgabezustand muss hierbei oftmals der Ausgabewert geschrieben werden. Die Reihenfolge des Schreibens von Wert und Zustand ist für die Weiterverarbeitung in anderen Zellen relevant. Wird zuerst der Zustand und anschließend der Wert einer Zelle geschrieben, so kann eine davon abhängige Zelle den Abschluss einer Berechnung erkennen, noch bevor der Ausgabewert gesetzt ist. Der eingelesene Wert wäre damit fehlerhaft.

Die Reihenfolge des Schreibens kann in einer Softwareimplementierung, auf Grund von Compileroptimierungen und Cachenutzung der Prozessoren, nur durch die Einführung zusätzlichen Kontrollstrukturen sichergestellt werden. Verschiedenen Lockmechanismen müssen hierfür implementiert werden, die wiederum die Auswertung des Zellularautomaten verlangsamen. Zur Implementierung eines solchen Verfahrens bedarf es einer genauen Untersuchung des Performanceunterschieds zwischen synchroner Kommunikation und asynchroner Kommunikation mit Locks, die im Rahmen dieser Arbeit nicht durchgeführt wurde.

3.6.4 Umsetzung der virtuellen Maschinen für CPU und GPU

Wie zu Beginn des Kapitels beschrieben, werden die virtuellen Maschinen von einem gemeinsamen Programm verwaltet. Die Zelltypen werden sowohl für die CPU als auch für die GPU aus den gleichen C++-Klassen instanziiert. Der Unterschied besteht lediglich in der Ausführung der virtuellen Maschinen und dem damit verbundenen Aufrufen der Update-Methoden der Zellen. Die Bezeichnung der Ausführungsumgebung als virtuelle Maschine beschreibt den Umstand, dass die Zellularautomaten nicht als eigenständiges Programm kompiliert werden. Die Zellularautomaten werden dynamisch aus den vorgegebenen Klassen und den generierten Modellen erzeugt. Die Verknüpfungen der Zellen untereinander wird mittels Zeigern auf ihre Speicherstellen realisiert.

Beim Start der virtuellen Maschinen werden die kompilierten Zellen inklusive ihrer Verknüpfungen zu anderen Zellen und eventueller Werte bzw. Bedingungen an die virtuellen Maschinen übergeben. Diese legen die Zellen in dem für sie nutzbaren Speicher (Arbeitsspeicher der CPU bzw. GPU) ab und passen die Zeiger der Zellen auf ihre Nachbarn und Werte entsprechend an. Das ist notwendig, da sich die Position der Zellen im Speicher auf den verschiedenen Architekturen unterscheidet und die Verknüpfungen der Zellen und damit ihre Funktionalität andernfalls nicht gewährleistet wäre.

Zur Auswertung der Zellularautomaten beinhalten die virtuellen Maschinen eine Methode, die so lange Updates des Zellularautomaten ausführt, bis die Hauptzelle (Einsprungspunkt des Programms: *main*-Funktion des Programms oder andere durch den Benutzer ausgewählte Funktion) ihre Berechnung abgeschlossen hat. Ein Update des Zellularautomaten bedeutet, das ein Aufruf

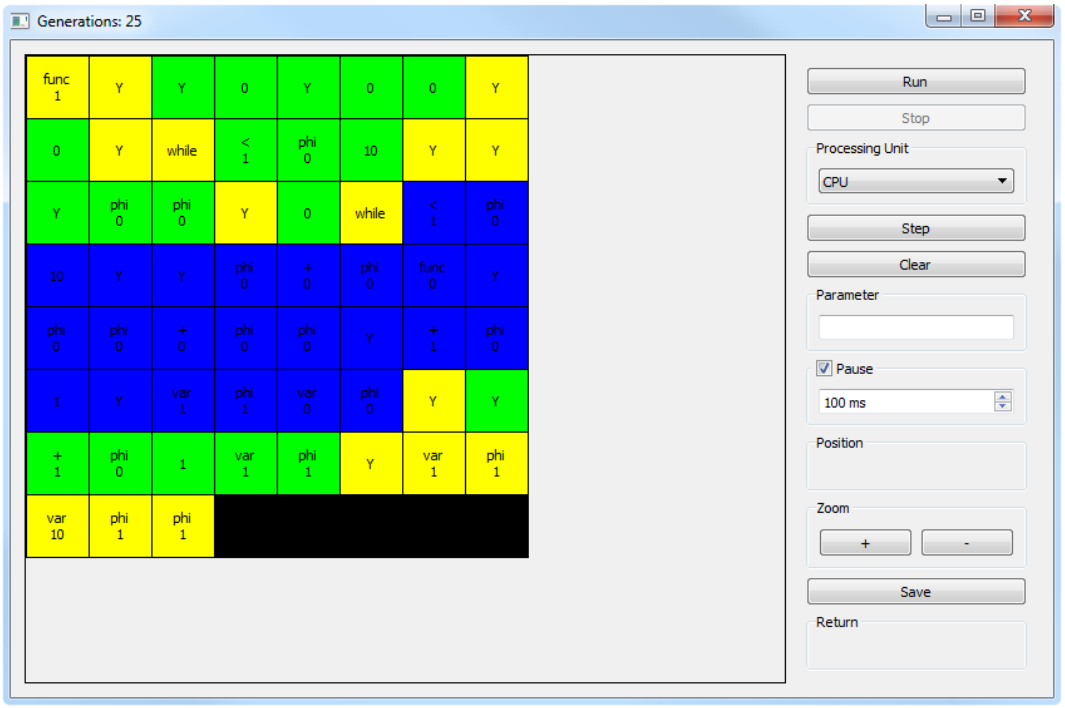


Abbildung 3.14: Auswertung eines Zellularautomaten im Debug-Modus

der Update-Methode aller enthaltenen Zellen ausgeführt wird.

Abbildung 3.14 zeigt die Oberfläche des Programms während der Ausführung eines Beispiels. Die Darstellung der einzelnen Zellen ist nur im Debug-Modus der virtuellen Maschine möglich. In diesem Modus kann jedes Update des Zellularautomaten einzeln durchgeführt werden. Die Zellen mit ihren aktuellen Werten und Zuständen werden dem Benutzer angezeigt. Dadurch ist es möglich, die Veränderungen innerhalb des Zellularautomaten in jedem Schritt bis zum Abschluss aller Berechnungen zu beobachten bzw. zu analysieren.

Die Darstellung der Zellen im Programm entspricht der in Kapitel 3.6.1 beschriebenen zweidimensionalen Gitterstruktur. Die unterschiedlichen Farben geben die in Kapitel 3.6.2 definierten Zustände wieder. Die Zuordnung der Farben zu den Zuständen ist:

- Schwarz $\widehat{=}$ *UNDEFINED/IDLE*,
- Gelb $\widehat{=}$ *COMPUTING*,
- Grün $\widehat{=}$ *EVAL_TRUE*,

- Rot $\widehat{=}$ *EVAL_FALSE*,
- Blau $\widehat{=}$ *CLEARING*,
- Grau $\widehat{=}$ *CLEARING_WR*.

Für die auf der CPU ausgeführte virtuelle Maschine werden die Updates des Zellularautomaten in zwei verschachtelten Schleifen durchgeführt. Die äußere Schleife läuft über die Spalten, die innere über die Zeilen des Zellularautomaten. Die parallele Ausführung und Verteilung der Zellen auf alle zur Verfügung stehenden Prozessoren wird durch *OpenMP*-Anweisungen [82] realisiert.

Im Debug-Modus der virtuellen Maschine, bei dem für den Benutzer die Möglichkeit besteht, alle Updates des Zellularautomaten einzeln zu durchlaufen, werden die Spalten des Zellularautomaten zu Blöcken zusammengefasst, die der Anzahl der verfügbaren Prozessoren entsprechenden. Um den Schrittbetrieb zu ermöglichen, werden bei jedem Update des Zellularautomaten neue Threads vom System angelegt/gestartet, die nach dem Update wieder gestoppt und zerstört werden müssen.

Die Performance ist in diesem Modus nicht relevant, dennoch werden die Zellupdates parallel ausgeführt, um ein möglichst nah am Release-Modus orientiertes Verhalten zu erreichen. Insbesondere betrifft das die in Kapitel 3.6.3 beschriebene asynchrone Zellkommunikation.

Führt der Benutzer die virtuelle Maschine im Release-Modus aus, so muss ein ständiges Anlegen und Zerstören von Threads vermieden werden, um die bestmögliche Performance zu erreichen. Beim Start der virtuellen Maschine werden hierzu einmalig Threads angelegt, die die Zellauswertung der ihnen zugewiesenen Zellen bis zum Abschluss aller Berechnungen übernehmen. Mittels Barrieren (`#pragma omp barrier`), die alle Threads erreichen müssen, bevor sie überschritten werden dürfen, wird der chronologisch korrekte Ablauf der Zellupdates über den gesamten Zellularautomaten sichergestellt.

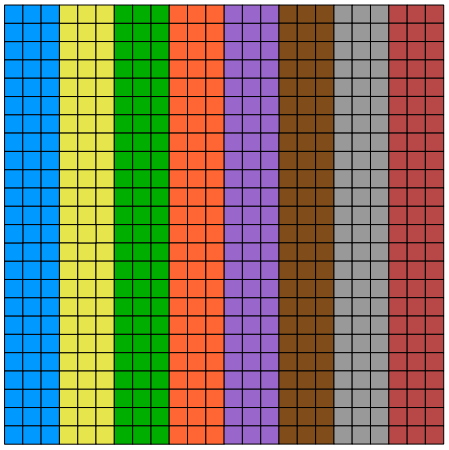
Nach jedem erfolgten Update des Zellularautomaten überprüfen die Threads, ob die Hauptzelle des Programms ihre Auswertung abgeschlossen hat. Ist das der Fall, so werden die Threads gestoppt und anschließend zerstört. Andernfalls starten sie die nächsten Updates der ihnen zugewiesenen Zellen. Ein Vergleich der Performance zwischen Debug-Modus und Release-Modus ist in Kapitel 5.2.1 angegeben.

Die GPU-Variante der virtuellen Maschine wurde mit dem von der *NVIDIA Corporation* bereitgestellten *CUDA Toolkit* [24] umgesetzt. Die virtuelle Maschine ruft bei der Auswertung des Zellularautomaten eine Kernel-Funktion auf, die ein Update einer Zelle des Zellularautomaten ausführt. Anders als auf der CPU wird die Kernel-Funktion jedoch nicht nur einmal gestartet. Sie erhält als Argument die Anzahl der zu verwendenden Threads und wird dementsprechend oft parallel ausgeführt. Die Aufteilung der Threads auf die zur Verfügung stehenden Prozessoren und das Pausieren bzw. Fortsetzen einzelner Threads wird vom GPU-internen Scheduler übernommen.

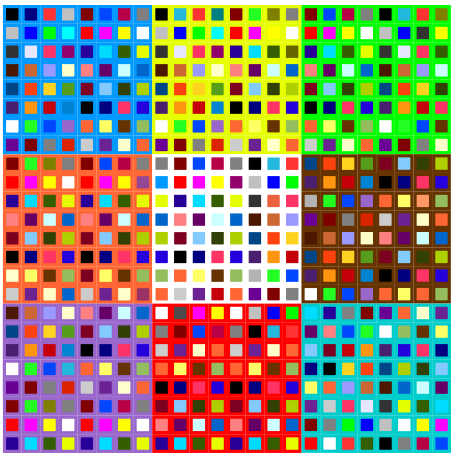
Threads auf der GPU sind leichtgewichtiger als Threads auf der CPU [29]. Damit erübrigt sich eine Kombination mehrerer Zellen innerhalb eines Threads, weil der Overhead der Threadverwaltung sehr klein ist und der Scheduler schnell zwischen den einzelnen Threads umschalten kann. Des Weiteren steht auf der GPU eine hohe Anzahl an Prozessoren zur Verfügung, von denen jeder mehrere, durch den Scheduler verwaltete, Threads verarbeiten kann (Beispiel: *NVIDIA GeForce 580 GTX* mit 512 Prozessoren und 48 Threads pro Prozessor).

Aufgrund von Architekturbeschränkungen müssen die Threads in Blöcke aufgeteilt werden, auf die vom Scheduler individuell zugegriffen werden kann und somit eine parallele Ausführung mehrerer Blöcke bzw. Threads innerhalb der Blöcke möglich ist. Eine Einschränkung bezüglich der Blockgröße ergibt sich aus der architekturspezifischen maximalen Anzahl an möglichen Threads pro Block. Eine andere Einschränkung resultiert aus dem pro Block nutzbaren gemeinsamen Speicher. Dieser ist relativ klein (Beispiel: *NVIDIA Geforce 580 GTX* mit 48KB pro Block), jedoch performanter als der globale Arbeitsspeicher der Grafikkarte. Er kann zur Beschleunigung der Zellauswertung verwendet werden, beschränkt dadurch aber zusätzlich die maximale Anzahl an Threads pro Block.

Zur Auswertung der Zellularautomaten werden zweidimensionale Thread-Blöcke mit 64 (8x8) Threads eingesetzt. Diese werden in einem zweidimensionalen Gitter, entsprechend der Anzahl der für den jeweiligen Zellularautomaten notwendigen Zellen, angeordnet und beim Aufruf der Kernel-Funktion als Parameter übergeben. Abbildung 3.15 stellt die unterschiedliche Verteilung der Zellen auf die Threads der CPU und GPU an einem Beispiel gegenüber.



(a) CPU mit acht Rechenkernen



(b) GPU mit 8x8 großen Threadblöcken individueller Threads

Abbildung 3.15: Unterschiedliche Aufteilung der Zellen eines Zellularautomaten mit 24x24 Zellen auf die Threads der CPU und GPU

Während der Ausführung der Kernel-Funktion werden jeweils 32 Threads vom Scheduler zu einem *Warp* zusammengefasst. Innerhalb eines Warps arbeiten alle Threads synchronisiert zueinander. Weichen Pfade der ausgeführten Algorithmen durch Verzweigungen oder Schleifen voneinander ab, so werden am Ende des alternativen Pfades Synchronisationspunkte bzw. Barrieren eingefügt. An diesen Barrieren müssen alle Threads ankommen, bevor diese überschritten und die Ausführung fortgesetzt werden kann. Dieses Verhalten stammt aus der ursprünglichen Verwendung von Grafikkarten zur Verarbeitung von Pixeln, bei denen alle Pixel die gleichen Verarbeitungsschritte durchlaufen müssen.

Die aufgeführten architekturspezifischen Eigenschaften der GPU führen zu unterschiedlich hohen Ausführungsgeschwindigkeiten je nach Wahl der Anzahl an Threads pro Block. Die verwendete Blockgröße von 64 Threads hat sich in verschiedenen Tests als optimale Aufteilung für die Implementierung der virtuellen Maschine zur Auswertung der Zellularautomaten auf der GPU herausgestellt. Ein Vergleich für unterschiedliche Konfigurationen ist in Kapitel 5.2.2 angegeben. Bedingt durch die Zusammenfassung von Threads zu Warps beeinflusst auch die Anordnung der Zellen innerhalb des Zellularautomaten die

Performance. Beispiele für den Einfluss der Zellanordnung sind in Kapitel 5.2.3 dargestellt.

Die Auswertung der Zellularautomaten kann wie bei der CPU-basierten Berechnung sowohl im Debug- als auch im Release-Modus ausgeführt werden. Im Debug-Modus findet nach jedem Update des Zellularautomaten ein Kopieren der Zellen in den Arbeitsspeicher der CPU und Anzeige der Zellzustände für den Benutzer statt (siehe Abbildung 3.14). Der zusätzliche Kommunikationsaufwand und das Starten des Update-Prozesses auf der GPU durch den Aufruf der Kernel-Funktion wird im Release-Modus optimiert durchgeführt. Ein Kopieren der Zellen wird, bis auf die Rückgabe der Ergebnisse des jeweiligen Algorithmus, nicht durchgeführt. Zusätzlich wird der Aufruf-Puffer der GPU zur Minimierung der Latenzzeit der Kernel-Aufrufe genutzt (siehe [15]). Die Kernel-Aufrufe werden asynchron ausgeführt, dass heißt, es wird nicht auf den Abschluss der Berechnungen gewartet. Lediglich nach jedem 100. Kernel-Aufruf wird ein Synchronisationspunkt eingefügt und der Zellularautomat auf der GPU auf mögliche Rückgabewerte überprüft. Eine experimentelle Untersuchung der Unterschiede zwischen Debug- und Release-Modus ist in Kapitel 5.2.1 angegeben.

Weitere Unterschiede zwischen Berechnungen auf der CPU und der GPU sowie verschiedene Optimierungsverfahren für GPU-Berechnungen sind in [25] beschrieben.

3.7 Hardwareimplementierung und Abbildung auf einen FPGA

Im Gegensatz zur softwarebasierten Auswertung werden beim hardwarebasierten Ansatz die Zellen nicht erst zur Laufzeit des Auswertungsprogrammes erzeugt. Die Zellen und ihre Verbindungen werden stattdessen vorab durch den Compiler in einer Hardwarebeschreibungssprache (VHDL) codiert und auf einen FPGA abgebildet. Anschließend werden die Zellen auf dem FPGA ausgewertet und die Ergebnisse an den Benutzer übermittelt.

Die Zellauswertung auf dem FPGA wird durch eine Netzwerkverbindung (Ethernet) zwischen dem FPGA und einem PC mit dem selben Programm, wie zur Verwaltung der virtuellen Maschinen für die CPU und GPU, gestartet und kontrolliert.

3.7.1 Codierung der Zellen und Kommunikationswege in VHDL

Für die Implementierung in Hardware wird ein Codegenerator eingesetzt, der den VHDL-Code für den Zellularautomaten erzeugt. Eine Gemeinsamkeit mit der Softwareimplementierung bei der Erzeugung der Zellen besteht in statisch implementierten Prototypen der verschiedenen Zelltypen. Dazu wurden in VHDL individuelle Module für jeden der in Kapitel 3.5.1 vorgestellten Zelltypen codiert. Zusätzlich wurde ein unvollständiges bzw. allgemeines Modul für den Zellularautomaten implementiert, das vom Codegenerator bei der Übersetzung in VHDL mit den spezifischen Daten des umzusetzenden Modells des jeweiligen Zellularautomaten komplettiert wird.

Die Aufgabe des Codegenerators besteht darin, den Instanziierungscode der Zellmodule inklusive aller *Port-Maps*, mittels derer die Verbindung der Zellen untereinander hergestellt wird, in das unvollständige Modul des Zellularautomaten einzufügen. Des Weiteren beschreibt der Codegenerator so genannte *Generics*, die Konstante Werte innerhalb der instanziierten Module darstellen und beispielsweise für Wertezellen benötigt werden.

Die Besonderheit des Verfahrens liegt darin, dass der Codegenerator nur eine Datei verändern muss, um jeden kompilierten Zellularautomaten in VHDL abzubilden. Die spezifischen Zellmodule der einzelnen Zelltypen müssen nicht verändert werden. Alle Anpassungen, die zur Übersetzung von Zellularautomaten in VHDL notwendig sind, können mittels Port- und Generic-Maps realisiert werden.

Die Updates der Zellen werden durch den, vom internen Taktgenerator des FPGAs, erzeugten Takt gesteuert. Ein Taktzyklus entspricht einem Update des kompletten Zellularautomaten. Das heißt, alle Zellen des Zellularautomaten werden innerhalb eines Taktzyklus parallel bearbeitet und entsprechend ihrer Zustände und Bedingungen ausgewertet.

Die an einen Takt gebundene Verarbeitung bietet einige Optimierungsmöglichkeiten, die in Software mit erheblich mehr Aufwand verbunden sind. Dadurch ist es beispielsweise möglich, den Overhead zur Synchronisation der Zellen untereinander zu reduzieren. Während in Software auf Grund von Compileroptimierungen sowie Cache- und Arbeitsspeicherverwaltung die Reihenfolge von Lese- und Schreiboperationen ohne Locks nicht garantiert werden kann, ist die Abarbeitung in Hardware fest durch den Takt vorgegeben. Alles was

innerhalb eines Taktzyklus passiert und als Signal in die Datenleitungen geschrieben wird, liegt im nächsten Takt an den damit verbunden Ports an. Auf einen zusätzlicher Puffer, wie er in der softwarebasierten Lösung (siehe Kapitel 3.6.2) eingesetzt wird, kann verzichtet werden.

Ein weiterer Vorteil gegenüber dem softwarebasierten Ansatz besteht in der Individualität der Zellen und ihre Verknüpfbarkeit untereinander. Während in Software alle Zellen aus einer Basisklasse abgeleitet werden und die gleichen Grundeigenschaften und Zustände haben, damit sie von der virtuellen Maschine verarbeitet werden können, ist das in Hardware nicht notwendig. Die Zelleigenschaften können so weit reduziert werden, dass lediglich ihr spezieller Anwendungsfall abgedeckt wird. Bei den meisten Zelltypen ist es ausreichend, zwischen den Zuständen „ausgewertet“ und „nicht ausgewertet“ zu unterscheiden. Das heißt, der Zustand der Zelle kann durch ein Bit dargestellt werden. Die Codierung des Zustandes durch ein einzelnes Bit kann im Gegensatz zur Softwareimplementierung, wo der kleinste Datentyp ein Byte umfasst, in VHDL dargestellt und implementiert werden.

Zellen, bei denen mehr als zwei Zustände notwendig sind, wie *if*-Zellen oder *while*-Zellen, können durch Hinzunahme eines weiteren Bits den Zustand *EVAL_TRUE* bzw. *EVAL_FALSE* anzeigen. Dieses 2-Bit breite Signal muss lediglich mit den davon abhängigen ϕ-Zellen verbunden werden. Für die Verbindung zu ihrer Elternzelle ist wiederum der Anteil des Signals, der zwischen „ausgewertet“ und „nicht ausgewertet“ entscheidet, ausreichend. Einsparungen im Bezug auf die Datenmenge sind durch den selektiven bitweisen Zugriff höher als in der Softwareimplementierung, wo dies nicht möglich ist.

Das *Reset*-Signal kann ebenfalls durch ein einzelnes Bit dargestellt werden, so dass zur Umsetzung der Basisfunktionalität der Zellen nur drei Bits notwendig sind:

1. *Clock* - eingehendes Takt-Signal des Zellularautomaten
   - 0 $\widehat{=}$ keine Verarbeitung
   - 1 $\widehat{=}$ Auswertung der Zelle
2. *Reset* - eingehendes Reset-Signal zur Zelle
   - 0 $\widehat{=}$ Zelle kann ausgewertet werden
   - 1 $\widehat{=}$ Zurücksetzen der Zelle

3. *State* - ausgehendes Zustands-Signal der Zelle
   - $0 \mathrel{\widehat{=}}$ „nicht ausgewertet“
   - $1 \mathrel{\widehat{=}}$ „ausgewertet“

Hinzu kommen je nach Zelltyp, neben dem bereits erwähnten zusätzlichen Zustandsbit, Signale zum Einlesen des Zustandes anderer Zellen sowie Signale zum Lesen und Schreiben von Werten.

3.7.2 Asynchrone Kommunikation und andere Optimierungen

Ähnlich der Verwendung von Pointern in Software zur Steuerung des Reset-Signals (siehe Kapitel 3.6.3) kann auch in Hardware die Zellhierarchie zur schnelleren Nachrichten- und Datenübertragung umgangen werden. Das Reset-Signal wird, wie in der Softwareimplementierung, durch Zellen, die Schleifen repräsentieren, generiert und ist direkt mit allen in der Schleife enthaltenen Zellen verbunden. Das erlaubt, die Schleife innerhalb eines Taktzyklus zurückzusetzen und für die nächste Iteration vorzubereiten.

Die Hardwareimplementierung ist nicht auf die einfache Weiterleitung von Nachrichten an viele Zellen beschränkt. Während in Software asynchrone Kommunikation und Verarbeitung nur begrenzt möglich ist bzw. Lock-Mechanismen zur korrekten Verwendung von Werten und Zuständen erforderlich macht, lässt sie sich in Hardware durch selbstsynchronisierende Verarbeitung realisieren.

Das kann beispielsweise für Operatorzellen ausgenutzt werden. Anstatt innerhalb eines taktgesteuerten Prozesses zu warten, bis die Kinderzellen ihren Zustand auf *EVAL_TRUE* gesetzt haben, um anschließend den eigenen Wert auf das Ergebnis der Operation und den eigenen Zustand auf *EVAL_TRUE* zu setzen, kann das gleiche Ergebnis durch (ungetaktete) Kombination der Eingangswerte erreicht werden. Der Zustand einer binären Operationszelle wird dadurch in dem gleichen Taktzyklus *EVAL_TRUE*, in dem die Zustands-Signale beider Kinderelemente den Zustand *EVAL_TRUE* beinhalten.

In Abbildung 3.16 wird der Unterschied zwischen taktsynchronisierter Verarbeitung innerhalb eines Prozesses und selbstsynchronisierender Verarbeitung ohne Prozess für ein Zellmodul, das eine binäre Operation (Addition) repräsentiert verdeutlicht. Das Beispiel zeigt eine mögliche Schreibweise in VHDL.

```
architecture ca_op_plus_arch of
    ca_op_plus is
begin

  process ( clk )
  begin

    if ( rising_edge( clk ) )
    then

      if ( reset = '0' and
        childState( 0 ) = '1' and
        childState( 1 ) = '1' )
      then
        state <= '1';
        v_out <= v_in( 0 )
          + v_in( 1 );
      else
        state <= '0';
      end if;

    end if;

  end process;
end ca_op_plus_arch;
```

(a) taktsynchronisierte Verarbeitung

```
architecture ca_op_plus_arch of
    ca_op_plus is
begin

  state <=
    '1' when reset = '0'
          and childState( 0 ) = '1'
          and childState( 1 ) = '1'
        else
    '0';

  v_out <= v_in( 0 ) + v_in( 1 );

end ca_op_plus_arch;
```

(b) selbstsynchronisierende Verarbeitung

Abbildung 3.16: Unterschied zwischen taktsynchronisierter und selbstsynchronisierender Verarbeitung am Beispiel einer Additions-Zelle

Ebenso besteht die Möglichkeit, selbstsynchronisierende Berechnungen innerhalb von Prozessen als auch taktsynchronisierte Anweisungen außerhalb von Prozessen durchzuführen.

Das Verfahren kann für verschiedene Zelltypen angewendet werden. Einerseits führt die selbstsynchronisierende Verarbeitung tendenziell zu einer komplexeren und langsameren Schaltung, da innerhalb eines Taktes mehr Operationen durchgeführt werden. Andererseits kann der so generierte VHDL-Code für die Zellularautomaten vom eingesetzte Synthesetool besser optimiert werden, da Zwischenergebnisse entfallen können. Ein experimenteller Vergleich zwischen taktsynchronisierter und selbstsynchronisierender Verarbeitung ist in Kapitel 5.3.1 angegeben.

Eine andere Optimierungstechnik betrifft die Kombination und Entfernung von Zellen. Während es in Software aufwändig ist, bestimmte Zellen zu entfernen und die Verweise auf diese Zellen korrekt umzuleiten, kann diese Art der Optimierung für die Hardwareimplementierung während der Codegenerierung erfolgen.

ϕ-Zellen, die ein einzelnes Argument beinhalten und keine zusätzlichen Bedingungen durch Schleifen berücksichtigen müssen, können bei der Codegenerierung durch Verweise auf das Argument selbst ersetzt werden. Das heißt, die von der ϕ-Zelle abhängige Elternzelle (beispielsweise eine Operatorzelle) erhält den Zustand und Wert mittels veränderter Port-Maps direkt vom Argument der ursprünglichen ϕ-Zelle, so dass diese entfernt bzw. nicht mit generiert werden muss.

Fork-Zellen sind in der Softwareimplementierung binäre Knotenpunkte, die Nachrichten an und von zwei Kinderzellen kombinieren und weiterleiten. Sind mehr als zwei Kinderzellen zu verwalten, werden *Fork*-Zellen hintereinander geschaltet. Eine beliebige Anzahl von Kinderzellen an einer einzelnen *Fork*-Zelle erfordert in der Softwareimplementierung eine variable Schleife (je nach Anzahl der Kinderzellen) zum Senden von Nachrichten bzw. zur Auswertung der Zustände der Kinderzellen. In der Hardwareimplementierung kann die Auswertung beliebig vieler (zum Zeitpunkt der Codegenerierung bekannter) Kinderzellen durch Port-Maps aller Zustände der Kinderzellen auf einen Vektor und Test des Vektors auf eine Eins an jeder Position realisiert werden. In VHDL-Code wird der Zustand von *Fork*-Zellen durch die Zuweisung:

```
state <= bool_to_logic( childStates = ( childStates'range => '1' ) )
```

definiert. Der Vergleich testet die Ausgabewerte des Vektors `childStates` gegen einen Vektor gleicher Länge, bei dem jedes Bit auf Eins gesetzt ist. Die Funktion `bool_to_logic` wandelt den booleschen Wahrheitswert (`true` bzw. `false`) des Vergleichstests in einen logischen Wert (**1** bzw. **0**) um. Der auf diese Weise ermittelte Wert wird anschließend dem Zustand der *Fork*-Zelle zugewiesen.

Die Anzahl an verwendeten Bits für die Übertragung von Werten zwischen Zellen bietet einen weiteren Ansatzpunkt für Optimierungen. Zum Beispiel haben sowohl *if*-Zellen als auch *while*-Zellen jeweils eine Kinderzelle, die ihre Bedingungen repräsentiert. Diese Kinderzelle kann einen beliebigen Datentyp

```
entity ca_if is
  port (
    clk       : in std_logic                        := '0';
    reset     : in std_logic                        := '0';

    lState    : in std_logic                        := '0';
    v_in      : in std_logic                        := '0';

    rState    : in std_logic                        := '0';

    state     : out std_logic_vector( 0 to 2 )      := "001" );
end ca_if;

architecture ca_if_arch of ca_if is
begin

  state( 0 ) <= ( lState and not v_in ) or rState; -- "ausgewertet"-Signal
  state( 1 ) <= lState and not v_in;               -- "EVAL_FALSE"-Signal
  state( 2 ) <= not ( lState and v_in );           -- "Reset"-Signal fuer
                                                   -- "then"-Block
end ca_if_arch;
```

Abbildung 3.17: *if*-Zelle in der finalen Implementierung

als Ausgabewert haben, muss bei der Verwendung als Bedingung jedoch in einen booleschen Wahrheitswert umgewandelt werden. Zur Vermeidung eines unnötig großen Wertebereiches bei der Übertragung sind Kinderzellen, die Bedingungen darstellen, dementsprechend angepasst. Ihre Ausgabewerte werden direkt vor dem Setzen der entsprechenden Signale in logische Werte umgewandelt. Dadurch muss ihr Ausgabe-Signal nur ein einzelnes Bit breit sein.

Kombiniert man die verschiedenen Verfahren miteinander, so ergeben sich zusätzliche Möglichkeiten zur Optimierung. Abbildung 3.17 zeigt die finale VHDL-Codierung der eingesetzten *if*-Zellen. Die Initialwerte der einzelnen Signale dienen zur Simulation der Zellen und haben keine Bedeutung für die FPGA-Implementierung.

Der Zustand einer *if*-Zelle wird mit drei Bits codiert:

- Das erste Bit beschreibt, ob die Zelle „ausgewertet“ oder „nicht ausgewertet“ ist. Für den Fall, dass die Bedingung der Zelle `v_in` erfüllt ist, entspricht das Bit dem Zustand des unter diesen Umständen auszuwertenden *then*-Blocks. Andernfalls entspricht das Bit dem Zustand der Zelle, die die Bedingung repräsentiert.

- Das zweite Bit dient der Auswertung des *EVAL_TRUE*/ *EVAL_FALSE*-Zustandes der `if`-Zelle durch davon abhängige ϕ-Zellen. Dieses Bit wird auf *EVAL_FALSE* ($\widehat{=}$ 1) gesetzt, wenn die Bedingungszelle ausgewertet und ihr Ergebnis `false` ist. In allen anderen Fällen ist es Null.

- Das dritte Bit beschreibt das Reset-Signal für den *then*-Block der *if*-Zelle. Ist die Bedingungszelle ausgewertet und ihr Ausgabewert ist `true`, dann wird das Reset-Signal auf Null gesetzt und die Zelle kann ausgewertet werden.

In der momentanen Architektur der `if`-Zellen werden weder das Takt-Signal `clk` noch das Reset-Signal `reset` verwendet. Das Takt-Signal wird aufgrund der selbstsynchronisierenden Verarbeitung nicht benötigt. Das Reset-Signal ist ebenfalls nicht notwendig, da die Ausgabe des Zustandes der `if`-Zelle durch die Weiterleitung und Kombination der Signale der Kinderzellen realisiert ist. Die Ports mit den entsprechenden Signalen werden durch die Optimierung des Synthesetools bei der Erzeugung der Hardware entfernt. Ihr Vorhandensein dient lediglich einer einfachen Austauschbarkeit der zu verwendenden Modul-Architektur.

3.7.3 Ausführung des Zellularautomaten auf einem FPGA und Kommunikation zum PC

Im Anschluss an die Codegenerierung kann der VHDL-Code durch ein Synthesetool in eine Beschreibung der für eine Hardwareimplementierung notwendigen Elemente umgewandelt werden. Diese Beschreibung wird durch weiterverarbeitende Programme auf die spezifische Architektur des zur Auswertung eingesetzten FPGAs abgebildet. Die durchführenden Verfahren werden als *Mapping* (Abbildung auf die zur Verfügung stehende Technologie) und *Place & Route* (Auswahl der Technologiebausteine und Verbindungswege zwischen den Komponenten) bezeichnet. Im Anschluss daran wird ein *Bit-File* generiert, das zur Programmierung bzw. Konfiguration des FPGAs verwendet wird. Weitere Einzelheiten zur Programmierung von FPGAs sind in [67] beschrieben.

Für die vorgestellten Schritte zur Umwandlung des VHDL-Codes in eine Hardwarebeschreibung und anschließende Abbildung auf einen FPGA wurde für die vorliegende Arbeit die von der Firma *Xilinx* entwickelte *ISE Design*

Suite 14.4 [115] verwendet. Bei dem eingesetzten FPGA handelt es sich um einen Virtex-5 XC5VLX110T auf einem Xilinx XUPV5-LX110T Entwicklungsboard [116]. Der vom Codegenerator erzeugte VHDL-Code beinhaltet jedoch keine architekturspezifischen Befehle oder Module. Dadurch können die generierten Zellularautomaten auch von anderen Synthesetools verarbeitet und auf beliebige FPGAs abgebildet werden.

Neben der Funktionalität des Zellularautomaten wird ein Zustandsautomat auf dem FPGA integriert, der die Kommunikation zwischen einem PC und dem FPGA ermöglicht. Die Verbindung erfolgt über die Ethernet-Schnittstelle des FPGAs bzw. PCs. Die Anbindung und der Zugriff findet mit Hilfe des von *Microsoft Research* zur Verfügung gestellten *Simple Interface for Reconfigurable Computing (SIRC)* [28] statt. Es enthält sowohl die notwendigen Module zur Implementierung auf dem FPGA als auch eine C++-Bibliothek zur Implementierung der Software für den PC.

Die C++-Bibliothek wurde in das Programm zur Kompilierung und Steuerung der Zellularautomaten integriert, das auch zur Verwaltung der virtuellen Maschinen eingesetzt wird. Der Datenaustausch zwischen PC und FPGA wird von der Software und dessen Benutzer initiiert. Anschließend wartet die Software auf den Abschluss der Auswertung auf dem FPGA.

Auf der Seite des FPGAs führt ein Verbindungsaufbau zu einem Signal des SIRC-Interfaces an den Zustandsautomaten, der den Zellularautomaten steuert und überwacht. Erhält der Zustandsautomat das Start-Signal wechselt er vom *Idle*-Zustand in einen *Run*-Zustand und startet daraufhin die Auswertung des Zellularautomaten. Das heißt, der Zustandsautomat aktiviert die Hauptzelle (kompilierte Hauptfunktion) des Zellularautomaten. Ist die Auswertung beendet, werden die Ausgabedaten des Zellularautomaten in einen Ausgabespeicher geschrieben und über die SIRC-Schnittstelle an den PC übermittelt. Die Software, die warten musste bis der FPGA seine Arbeit abgeschlossen hat, kann anschließend die Ergebnisse einlesen und für den Benutzer darstellen.

Abbildung 3.18 zeigt den benutzergesteuerten Ablauf der Kommunikation zwischen PC und FPGA bei der Auswertung bzw. beim Zurücksetzen eines Zellularautomaten.

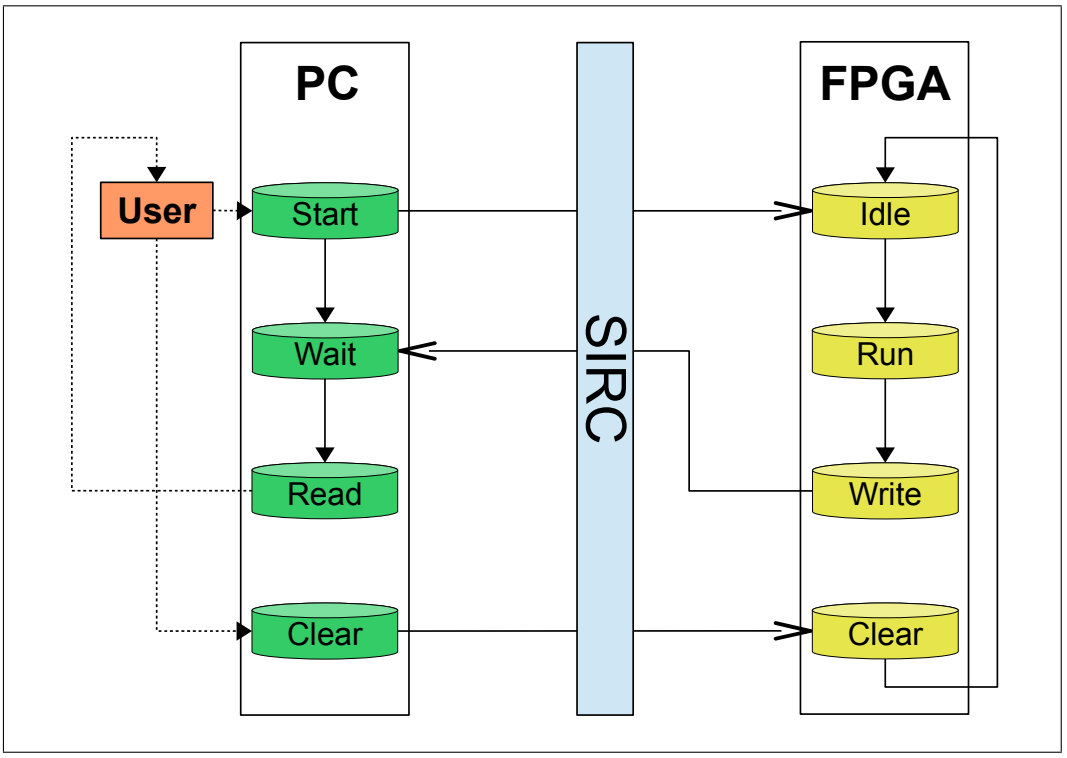


Abbildung 3.18: Benutzergesteuerte Kommunikation und Zustandswechsel zwischen PC und FPGA

4 Erweiterungen des Basis-Systems

Die bisher beschriebenen Zellularautomaten sind in ihrer Funktionalität und der Abbildbarkeit von Algorithmen auf die Zellularautomaten stark eingeschränkt. Sie enthalten beispielsweise keine Zellen zur Darstellung von Arrays und Pointern. Daneben ist auch der Datentyp der in Kapitel 3 beschriebenen Zellen auf Integer-Werte beschränkt.

Dieses Kapitel befasst sich mit Erweiterungen des Compilers und der Laufzeitumgebung, um die aufgeführten Elemente und Konzepte auf Zellularautomaten abzubilden. Dadurch soll vor allem die Idee der automatischen parallelen Ausführung sequentieller Programme für komplexe Probleme und Algorithmen ermöglicht werden.

4.1 Kompilierung von Arrays

Arrays bilden einen fundamentalen Bestandteil der C-Programmierung. Sie werden für unterschiedliche Probleme eingesetzt, zum Beispiel zur Sortierung von Daten, für mathematische Berechnungen mit Vektoren und Matrizen sowie in der Audio- und Videobearbeitung. Ohne die Möglichkeit der Speicherung gleichartiger Variablen in einem Array sind Algorithmen für die genannten Anwendungsfälle schwer umsetzbar.

Für einen Sortieralgorithmus müsste beim Einlesen und Sortieren von zehn Zahlen jedem Wert eine einzelne Variable zugewiesen werden, um sie anschließend gegeneinander zu vergleichen. Für den Programmierer erlaubt die Nutzung von Arrays eine vereinfachte Implementierung, die mit Schleifen umgesetzt werden kann und unter anderem beim *Bubblesort*-Algorithmus [69] eingesetzt wird.

Aus den genannten Gründen wurde für diese Arbeit ein Verfahren entwickelt, Arrays in das Modell der in Kapitel 3 vorgestellten Zellularautomaten zu übertragen. Nachfolgend werden Verfahren zur Abbildung von Arrays mit konstanten und variablen Elementzugriffen sowie die Vorverarbeitung mehrdimensionaler Arrays durch den Präcompiler beschrieben.

4.1.1 Erweiterung des Präcompilers zur Eliminierung mehrdimensionaler Arrays

Ein erster Schritt, der die Erweiterung des Compilers und der Laufzeitumgebung um Arrays ermöglicht, ist eine Reduktion mehrdimensionaler Arrays auf eine Dimension. Diese Aufgabe übernimmt der in Kapitel 3.3 vorgestellte Präcompiler. Die Reduktion von Arrays auf eine Dimension führt zu einer vereinfachten Datenabhängigkeitsanalyse, da anstatt mehrerer, nur noch einzelne Indizes bei Lese- und Schreibzugriffen auf Arrayelemente betrachtet und verglichen werden müssen.

Bei der Abbildung eines mehrdimensionalen auf ein eindimensionales Array muss zuerst ein neues Array angelegt werden, das in seiner Größe dem Produkt aller Dimensionen des ursprünglichen Arrays entspricht. Anschließend überprüft der Präcompiler alle im Programm vorkommenden Lese- und Schreibzugriffe auf die einzelnen Indizes des Arrays und rechnet sie auf die neuen Positionen der Elemente um.

Der neue Index idx' eines Arrayelementes ergibt sich bei einem n-dimensionalen Array aus den ursprünglichen Indizes $idx_1 \ldots idx_n$ und den zugehörigen Dimensionengrößen $dim_1 \ldots dim_n$ nach folgender Formel:

$$\begin{aligned} idx' &= (idx_1 \cdot dim_2 \cdot \ldots \cdot dim_n) \\ &+ (idx_2 \cdot dim_3 \cdot \ldots \cdot dim_n) \\ &\vdots \\ &+ (idx_n) \ . \end{aligned}$$

Der Zugriff auf ein dreidimensionales Array und die auf eine Dimension reduzierte Variante ist in Abbildung 4.1 visualisiert.

4.1.2 Arrays mit konstanten Indizes

Bei der Verwendung von Arrays ist die Art des Zugriffs auf die einzelnen Elemente relevant für die Datenabhängigkeitsanalyse. Die erste Zugriffsart ist ein statischer Zugriff mit bereits zum Kompilierungszeitpunkt bekannten Indizes.

Wird in dem gesamten zu kompilierenden Algorithmus lediglich statisch auf ein Array zugegriffen, so kann jedem Zugriff exakt ein Arrayelement zugeordnet werden. Die eindeutige Zuordnung der Zugriffe zu den Elementen hat den Vor-

```
#define D1 4
#define D2 3
#define D3 2

int a[ D1 ][ D2 ][ D3 ];

...

// Summe ueber alle Elemente
for ( i = 0; i < D1; i++ )
{
  for ( j = 0; j < D2; j++ )
  {
    for ( k = 0; k < D3; k++ )
    {
      summe += a[ i ][ j ][ k ];
    }
  }
}
```

```
#define D1 4
#define D2 3
#define D3 2

int a[ D1 * D2 * D3 ];

...

// Summe ueber alle Elemente
for ( i = 0; i < D1; i++ )
{
  for ( j = 0; j < D2; j++ )
  {
    for ( k = 0; k < D3; k++ )
    {
      summe += a[ ( i * D2 * D3 )
        + ( j * D3 ) + k ];
    }
  }
}
```

Abbildung 4.1: Reduktion eines dreidimensionalen Arrays auf eine Dimension und Anpassung der Zugriffe

teil, dass eine explizite Datenabhängigkeitsanalyse durchgeführt werden kann. Dadurch können die einzelnen Elemente des Arrays bei der Ausführung des Zellularautomaten getrennt behandelt und ohne Synchronisation untereinander gelesen bzw. geschrieben werden. Die Elemente des Arrays werden in diesem Fall wie individuelle Variablen behandelt und ihre parallele Bearbeitung führt zu einem performanten Zellularautomaten.

Bei der Überführung eines Algorithmus in die SSA-Darstellung (siehe Kapitel 3.4.1) wird jeder Zugriff auf eine generierte Variable einer ursprünglichen Variable zugeordnet, um die Datenabhängigkeiten zu protokollieren. Dieses Verhalten des Compilers wurde bei der Erweiterung um Arrays dahingehend verändert, dass jeder Zugriff nicht einer einzelnen Variable, sondern einem Paar aus Variable und Index zugeordnet wird. Zugriffe auf einzelne Variablen erhalten in der neuen Form eine Zuordnung mit dem Index Null. Somit muss im Compiler nicht zwischen einzelnen Variablen und Arrays differenziert werden. Variablen werden vom Compiler als Arrays mit einem Element behandelt.

Sind alle Indizes eines Arrays zum Kompilierungszeitpunkt berechenbar, so entsteht am Ende der Datenabhängigkeitsanalyse eine Liste mit allen Lese- und Schreibzugriffen sowie den zugehörigen Indizes des Arrays. Die anschließend einzufügenden ϕ-Funktionen erhalten als Argumente lediglich die Elemente der Liste, die auf Grund ihrer Indizes berücksichtigt werden müssen. Eine Unterscheidung zur Laufzeit des Zellularautomaten und Anpassung der implementierten ϕ-Zellen ist bei Arrays mit konstanten Indizes durch das beschriebene Verfahren unnötig.

4.1.3 Arrays mit variablen Indizes

Enthält der vom Programmierer entworfene Algorithmus Arrays mit variablen Indizes, so kann das in Kapitel 4.1.2 beschriebene Verfahren nicht bzw. nur teilweise angewendet werden. Bei der Protokollierung der Zuordnungen von Schreibe- und Lesezugriffen auf die Elemente eines Arrays erhalten alle variablen, und damit zum Kompilierungszeitpunkt nicht auswertbaren, Indizes eine Zuordnung zum Wert [-1].

Erkennt die Datenabhängigkeitsanalyse einen entsprechend gekennzeichneten Lesezugriff, so überprüft sie, ob ein Synchronisationspunkt vor dem Zugriff eingefügt werden muss. Synchronisationspunkte sorgen für einen einheitlichen Zustand eines Arrays, der als Ausgangspunkt für alle nachfolgenden Operationen auf das Array genutzt wird. Ein neuer Synchronisationspunkt muss erstellt werden, wenn zwischen dem letzten Synchronisationspunkt und dem aktuellen Lesezugriff Schreibzugriffe mit konstanten Indizes stattgefunden haben. Diese werden zur Maximierung der Performance eingesetzt (siehe Kapitel 4.1.2), soweit es die Datenabhängigkeiten erlauben. Zur korrekten Ausführung der auf Zellularautomaten abgebildeten Algorithmen müssen bei einem Lesezugriff mit variablem Index alle Elemente des zugehörigen Arrays berücksichtigt werden, aus denen zur Laufzeit das korrekte Element ausgewählt wird. Das macht einen Synchronisationspunkt vor dem Lesezugriff unumgänglich.

Bei einem Lesezugriff mit konstantem Index kann entweder auf den letzten Synchronisationspunkt (sofern dieser vorhanden ist) oder auf das vorab durch einen Schreibzugriff mit konstantem Index erzeugte Element zugegriffen werden. Schreibzugriffe mit variablem Index erzeugen automatische Synchronisationspunkte, die in der Datenabhängigkeitsanalyse berücksichtigt werden.

```
1  func( int idx )
2  {
3    int A[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
       };
4
5    int i = 0;
6    int x = 0;
7    int y = 0;
8
9    for ( i = 0; i < idx; i++ )
10     A[ i ] = 0;
11
12   // Synchronisationspunkt
13   // A = A
14
15   x = A[ 0 ] + A[ 1 ];
16   y = A[ 2 ] + A[ 3 ];
17
18   return x * y;
19 }
```

Abbildung 4.2: Einfügen von Synchronisationspunkten zur kontrollierten Steuerung von Arrayzugriffen und Erhaltung der Datenparallelität

Das individuelle Einfügen von Synchronisationspunkten hat den Vorteil, dass die unabhängige und dadurch parallele Auswertung der Zellen weitestgehend stattfinden kann. Einzig für den Fall, dass eine fehlerfreie Bearbeitung nicht garantiert werden kann, werden Synchronisationspunkte eingefügt. Diese führen dazu, dass im Anschluss an die Synchronisation die Zellen wieder parallel ausgewertet werden können. Ohne die Verwendung von Synchronisationspunkten müsste ab dem Punkt im Algorithmus, an dem ein Zugriff mit variablem Index stattgefunden hat, sequentiell auf die Zellen zugegriffen werden, da die Datenabhängigkeiten zum Kompilierungszeitpunkt unbekannt sind.

Abbildung 4.2 zeigt einen Algorithmus, bei dem ein Synchronisationspunkt nach einer Schleife eingefügt wird, um anschließend wieder parallel arbeiten zu können. Die Formel im Kommentar in Codezeile 13 ist keine korrekte C-Syntax. Sie beschreibt vielmehr die Berechnung innerhalb der Synchronisationszelle im kompilierten Zellularautomaten. Die Zelle enthält ϕ-Funktionen für jedes Element des Arrays und setzt die Werte anhand der in der Datenabhängigkeits-

analyse festgelegten Argumente und zur Laufzeit ausgewerteten Bedingungen. Im Beispiel kann jedes Element des Arrays am Synchronisationspunkt entweder den Startwert des Elements oder den innerhalb der Schleife gesetzten Wert erhalten. Das ist abhängig vom Eingabeparameter `idx` der Funktion `func`.

Ist die Auswertung der Synchronisationszelle abgeschlossen, können die Werte von `x` und `y` vollständig parallel berechnet werden (siehe Zellauswertung in Kapitel 3.5.2), da sie unabhängig voneinander sind.

Die variablen Zugriffe auf Elemente eines Arrays bzw. der Synchronisationszellen erfolgen durch erweiterte ϕ-Zellen. Diese haben neben ihrer ursprünglichen Funktion ein Kinderelement, das den variablen Arrayindex darstellt. Bei der Auswertung der ϕ-Zelle wird zuerst die Berechnung des Indexes gestartet. Ist der Index ermittelt und an die ϕ-Zelle übertragen, beginnt diese ihre Auswertung und überprüft die Argumente und Bedingungen auf Gültigkeit. Anschließend wird der Wert des Arguments an der dem Index entsprechenden Stelle ausgelesen und somit die Auswertung der ϕ-Zelle beendet.

Die Use-Conditions einer solchen ϕ-Zelle unterscheiden sich teilweise von den in Kapitel 3.4.1 vorgestellten. Außer der Pfad-Bedingung zur Anwendung des Arguments beinhalten sie zusätzlich eine Überprüfung des Indexes. Das trifft nicht für Zugriffe auf Synchronisationszellen zu, da sie jedes Element des Arrays enthalten und dadurch keine Einschränkungen beim Zugriff entstehen. Vielmehr werden sie verwendet, um die Gleichheit der Indizes zwischen ϕ-Funktionen und durch variable Schreibzugriffe entstandene Argumente zu überprüfen. Stimmen die Indizes nicht überein, so ist die Bedingung für das Argument nicht erfüllt. Es handelt sich in diesem Fall um ein potenzielles Argument, das erst durch die Validierung des Indexes zur Laufzeit zugelassen oder ausgeschlossen wird.

Die Überprüfung von Indizes kann auch durch zwei Synchronisationspunkte, einem vor dem Schreiben des Elementes sowie einem danach, umgangen werden. Da jedoch jeder Synchronisationspunkt aus einer vollständigen Kopie des Arrays besteht und zusätzlich ϕ-Zellen zum Einlesen der Elemente notwendig sind, wird aus Optimierungsgründen, bezüglich der Größe des erzeugten Programms, das Verfahren für den in dieser Arbeit beschriebenen Compiler nicht eingesetzt.

```
int func( int i, int j )
{
  int A[] = { 1, 2, 3 };

  A[ i ] = 4;

  return A[ j ];
}
```

(a) Originaler C-Code

```
int func( int i_0, int j_0 )
{
  int A_0[] = { 1, 2, 3 };

  A_1[ phi_0( i_0 ) ] = 4;

  return phi_3( A_0, A_1 )
    [ phi_2( j_0 ) ];
}
```

(b) SSA-Darstellung

Abbildung 4.3: Einführung von ϕ-Funktionen mit Indizes

Das in Abbildung 4.3 dargestellte Beispiel zeigt die Verwendung von ϕ-Funktionen mit Indizes. Die Bezeichnung `A_1` entspricht der Verwendung in der Datenabhängigkeitsanalyse. Dabei wird das Element wie ein vollständiges Array berücksichtigt und als Argument von `phi_3` hinzugefügt. Lediglich in der internen Implementierung der Zellen und Bedingungen wird nach dem oben beschriebenen Verfahren zwischen einem vollständigen Array und einem variablen Element innerhalb des Arrays differenziert und das Argument bei Ungleichheit des Indexes zur Laufzeit verworfen.

4.2 Übersetzung von Pointern

Pointer stellen eine Besonderheit der C-Programmierung dar. Während in anderen Programmiersprachen, zum Beispiel in Java, implizit Referenzen von Objekten verwendet werden, hat in C der Programmierer die explizite Kontrolle über die Daten. Der Entwickler entscheidet, ob der Wert einer Variablen oder eine Referenz auf die Variable, in Form eines Pointers auf die Adresse der Variable, verwendet wird. Dadurch ist unter anderem die Programmierung von *Call-by-Value* Funktionsaufrufen, bei denen die Werte an die Funktionsargumente kopiert werden, sowie von *Call-by-Reference* Funktionsaufrufen, bei denen die Adressen der Aufrufparameter übergeben werden, möglich (weitere Informationen zur Parameterübergabe in C sind in [65] angegeben).

Die bisher vorgestellten Verfahren und Konzepte erlauben lediglich die Verwendung von *Call-by-Value* Funktionsaufrufen, da keine Zellen zur Repräsentation von Pointern existieren. In diesem Kapitel wird ein Verfahren zur Er-

weiterung der vorgestellten Zellularautomaten um Pointer, notwendige algorithmische Vorverarbeitungen sowie die Erkennung von Datenabhängigkeiten beschrieben.

4.2.1 Vorverarbeitung von Pointern durch den Präcompiler

Der Einsatz von Pointern dient hauptsächlich der Realisierung von *Call-by-Reference* Funktionsaufrufen, die beispielsweise genutzt werden, um ein Array innerhalb einer Funktion mit Werten zu füllen. Anschließend kann das gefüllte Array in anderen Funktionen weiterverarbeitet werden.

Aufgabe des Präcompilers ist es, alle Funktionsaufrufe in eine einheitlich Form zu transformieren, so dass die Datenabhängigkeitsanalyse nach einem für alle Funktionsaufrufe mit Pointern gemeinsamen Schema ablaufen kann. Jedes Aufrufargument, das als Referenz an die zugehörige Funktion übergeben werden soll, muss für die implementierte Datenabhängigkeitsanalyse aus genau einem Pointer bestehen. Um das zu erreichen, werden alle Aufrufargumente, die über den Adressoperator auf eine Variable zugreifen, durch eine Hilfsvariable bzw. einen Hilfspointer ersetzt.

Die Hilfsvariable wird als neue lokale Variable innerhalb der den Aufruf beinhaltenden Funktion eingefügt. Die Adresszuweisung des ursprünglichen Arguments wird vom Präcompiler direkt vor den Funktionsaufruf verschoben. Abbildung 4.4 zeigt das Vorgehen an einem Beispiel. Codezeile 12 in Abbildung 4.4(a) enthält den originalen Funktionsaufruf mit dem auf Variable `x` angewendeten Adressoperator. Dieser ist in Abbildung 4.4(b) vor den Funktionsaufruf verschoben (Codezeile 11) und die Adresse von `x` wird der Hilfsvariable `x_p` zugewiesen. In Codezeile 12 wird anschließend der Funktionsaufruf mit der Hilfsvariable `x_p` als Argument durchgeführt.

In einem weiteren Bearbeitungsschritt, nach dem Einfügen aller notwendigen Hilfsvariablen, ergänzt der Präcompiler alle Aufrufargumente, die Pointer enthalten, um ein zusätzliches Initialisierungsargument. Die Initialisierungsargumente werden in der auf den Präcompiler folgenden Datenabhängigkeitsanalyse durch ϕ-Funktionen ersetzt, die als Argumente alle SSA-Instanzen der Variable enthalten, auf die der jeweilige Pointer zeigen kann. Das dient der korrekten Einsetzung der Werte zur Laufzeit des Zellularautomaten. Die Pointer selbst werden lediglich für die Datenabhängigkeitsanalyse verwendet.

```
1  void func( int* a_p )
2  {
3    ...
4  }
5
6  int main()
7  {
8    int x = 5;
9
10
11
12   func( &x );
13
14   return 0;
15 }
```

(a) Originaler C-Code

```
1  void func( int* a_p )
2  {
3    ...
4  }
5
6  int main()
7  {
8    int x = 5;
9    int* x_p;
10
11   x_p = &x;
12   func( x_p );
13
14   return 0;
15 }
```

(b) C-Code mit eingefügter Hilfsvariable

Abbildung 4.4: Einfügen von Hilfspointern in Funktionsaufrufe

4.2.2 Zuweisungen an dereferenzierte Pointer in der SSA-Darstellung

Pointer stellen einen Vorteil für den Programmierer im Bezug zur Mächtigkeit der damit umsetzbaren Algorithmen dar, führen jedoch zu Problemen bei der Datenabhängigkeitsanalyse für die parallele Ausführung der Zellularautomaten. Bei der Umwandlung eines Algorithmus in die SSA-Darstellung (siehe Kapitel 3.4.1) reicht es beim Lesen bzw. Schreiben von Variablen über dereferenzierte Pointer nicht aus, die möglichen Instanzen einer einzelnen Variable zu betrachten. Es müssen die SSA-Instanzen aller Variablen betrachtet werden, deren Adressen im Pointer hinterlegt sein können.

Ein Pointer selbst hat an der jeweiligen Programmstelle jedoch auch mehrere mögliche SSA-Instanzen. Die bei Lesezugriffen über ϕ-Funktionen potenziell infrage kommenden Werte sind somit von den an der betrachteten Programmstelle gültigen SSA-Instanzen der Pointer-Variable abhängig, die eventuell auf verschiedene Variablen zeigen, von denen wiederum mehrere SSA-Instanzen gültig sein können.

Abbildung 4.5 verdeutlicht das Problem an einem Beispiel. Die Zuweisung der Adresse an die Pointer-Variable `p` in Codezeile 12 bzw. 14 kann direkt in die SSA-Darstellung transformiert werden. Ebenso ist es möglich, die lesenden Zugriffe durch die ϕ-Funktionen `phi_2` bzw. `phi_3` in die SSA-Darstellung zu

```
1  int func()
2  {
3    int* p;
4
5    int x = 5;
6    int y = 3;
7
8
9
10
11   if ( ... )
12     p = &x;
13   else
14     p = &y;
15
16   (*p)++;
17
18
19   return x;
20 }
```

(a) Originaler C-Code

```
1  int func()
2  {
3    int* p_0, p_1;
4
5    int x_0, x_1;
6    int y_0, y_1;
7
8    x_0 = 5;
9    y_0 = 3;
10
11   if ( ... )
12     p_0 = &phi_0( x_0 );
13   else
14     p_1 = &phi_1( y_0 );
15
16   [ x_1, y_1 ]? = phi_3( *phi_2(
17     p_0, p_1 ) ) + 1;
18
19   return phi_4( x_0, x_1 );
20 }
```

(b) SSA-Darstellung

Abbildung 4.5: SSA-Darstellung von Pointern und auftretende Probleme

übertragen. Der schreibende Zugriff in Codezeile 16 an den dereferenzierten Pointer `*p` kann jedoch nicht mit den bisher vorgestellten Methoden in der SSA-Darstellung abgebildet werden.

Eine Möglichkeit zur Lösung des Problems ist in [71] beschrieben. Die Autoren nutzen eine Nummerierung innerhalb der SSA-Darstellung, um bestimmte Teile des Codes zu betrachten bzw. auszublenden. Die Auswahl der korrekten Variable wird zur Laufzeit in Abhängigkeit der Zuweisungen an die jeweiligen Pointer festgelegt. Die Zuweisungen bestimmen somit die jeweils gültige SSA-Nummer. In [43] wird ein ähnliches Verfahren angewendet, das anstatt einer Nummerierung eine *points-to*-Menge für jeden Pointer berücksichtigt.

Im für die vorliegende Arbeit entwickelten System wird ebenfalls eine *points-to*-Menge für die Erkennung der Datenabhängigkeiten von dereferenzierten Pointern eingesetzt. In der momentanen Version beschränkt sich diese auf eine Variable. Das heißt, jedem Pointer darf exakt einmal innerhalb eines Programms eine Adresse zugewiesen werden. Eine Erweiterung der *points-to*-Menge um mehrfache Zuweisungen von Adressen an Pointer könnte in zukünftige Weiterentwicklungen des Systems einfließen.

4.2.3 Datenabhängigkeitsanalyse mit Pointern

Die für die vorliegende Arbeit implementierte Datenabhängigkeitsanalyse zur Berücksichtigung von Pointern ist eine Erweiterung des in Kapitel 3.4.1 vorgestellten Verfahrens um eine points-to-Menge (siehe Kapitel 4.2.2) für jede Pointer-Variable. Diese Menge wird bei Zuweisungen und ϕ-Funktionen, die von der Pointer-Variable abhängig sind, berücksichtigt. Dabei müssen die folgenden Fälle unterschieden werden:

1. Zuweisung einer Adresse an eine Pointer-Variable
   - Beispiel: {`int* x_p = &x;`} bzw. {`int* x_p = &phi_0( x );`} (in SSA-Darstellung)
   - Update der *points-to*-Menge des Pointers (darf aufgrund der Einschränkungen in Kapitel 4.2.2 exakt einmal im zu kompilierenden Algorithmus geschehen)
2. Zuweisung eines Pointers an einen anderen Pointer
   - Beispiel: {`int* y_p = x_p;`} bzw. {`int* y_p = phi_0( x_p );`} (in SSA-Darstellung)
   - Kopieren der *points-to*-Menge des zugewiesenen Pointers (beide Pointer zeigen anschließend auf die selbe Variable)
3. Zuweisung an einen dereferenzierten Pointer
   - Beispiel: {`*x_p = 5;`}
   - Update der SSA-Menge der Variable, auf die der Pointer zeigt (Ersetzung der vorhandenen Instanzen durch den dereferenzierten Pointer)
   - entspricht dem Einfügen einer neuen SSA-Instanz der Variable, auf die der Pointer zeigt (siehe Kapitel 3.4.1 bzw. [79])
4. Lesen eines dereferenzierten Pointers
   - Beispiel: {`int y = *x_p;`} bzw. {`int y = phi_1( *phi_0( x_p ) );`} (in SSA-Darstellung)
   - Einsetzen aller (an der Stelle des Algorithmus) SSA-Instanzen der Variable, auf die der Pointer zeigt (vorheriges Auslesen der *points-to*-Menge notwendig)

Wie in Kapitel 4.2.1 beschrieben, müssen Funktionsaufrufe mit Pointer-Argumenten gesondert behandelt werden. Im Gegensatz zu Funktionsaufrufen ohne Pointer-Argumente ist es notwendig, die Veränderung der über Pointer übergebenen Variablen innerhalb der aufgerufenen Funktion in der Datenabhängigkeitsanalyse zu berücksichtigen. Die eventuell erzeugten neuen SSA-Instanzen der jeweiligen Variable sind auch nach Rückkehr aus der Funktion gültig.

Um die in Funktionen erzeugten SSA-Instanzen der Variablen in anschließende Programmteile einzubeziehen, wird in der implementierten Datenabhängigkeitsanalyse ein aus verteilten Systemen bekanntes Verfahren zur Parameterübergabe genutzt. Das so genannte *Call-by-Value-Result* Verfahren [100] dient in verteilten Systemen der Realisierung von *Call-by-Reference* Funktionsaufrufen über getrennte Adressräume. Die Parameter werden als Werte an die aufgerufene Funktion übermittelt und die veränderten Werte nach Beendigung der Funktion wieder an den Aufrufer zurück übertragen. Danach erfolgt die Ersetzung der vor dem Funktionsaufruf gültigen Werte der Variablen durch die von der Funktion erhaltenen.

In der Datenabhängigkeitsanalyse werden auf die gleiche Art und Weise die gültigen SSA-Instanzen einer Variable bei einem Funktionsaufruf eingesetzt und anschließend durch die am Ende der Funktion gültigen SSA-Instanzen ersetzt. Das Einsetzen der SSA-Instanzen der Variable beim Aufruf der Funktion zur Laufzeit geschieht mithilfe der in Kapitel 4.2.1 beschriebenen Initialisierungsargumente.

Abbildung 4.6 zeigt das Vorgehen der implementierten Datenabhängigkeitsanalyse an einem Beispiel. In der SSA-Darstellung in Codezeile 1 ist die Ergänzung der Funktion um ein zusätzliches Initialisierungselement erkennbar, das zur Laufzeit des Programms durch den Aufrufparameter `phi_2( x_0 )` (in Codezeile 18) gesetzt wird. Der Rückgabewert der Funktion wird durch die ϕ-Funktion `phi_3` gebildet. Die an dieser Stelle möglichen SSA-Instanzen der ursprünglichen Variable `x` können dabei sowohl `x_0` als auch die von der bedingten Ausführung innerhalb der Funktion `func` abhängigen dereferenzierten Pointer `*a_p_1` und `*a_p_2` sein.

Das beschriebene Verfahren macht eine Erweiterung der Zellularautomaten um zusätzliche Zelltypen für Pointer unnötig. Alle zur Ergänzung des Systems

```
1  void func( int* a_p )
2  {
3
4
5    if ( ... )
6      *a_p = 1;
7    else if ( ... )
8      *a_p = 2;
9  }
10
11
12 int main()
13 {
14   int x = 0;
15   int* x_p = &x;
16
17   func( x_p );
18
19
20   ...
21
22   return x;
23
24 }
```

(a) Originaler C-Code

```
1  void func( int* a_p_0 = &a_0 )
2  {
3    int* a_p_1, a_p_2;
4
5    if ( ... )
6      *a_p_1 = 1;
7    else if ( ... )
8      *a_p_2 = 2;
9  }
10
11
12 int main()
13 {
14   int x_0 = 0;
15   int* x_p_0 = &phi_0( x_0 );
16
17   func( phi_1( x_p_0 ){
18     phi_2( x_0 ) } );
19
20   ...
21
22   return phi_3( x_0,
23     *phi_3( a_p_1, a_p_2 ) );
24 }
```

(b) SSA-Darstellung

Abbildung 4.6: Datenabhängigkeitsanalyse mit Pointern

um Pointer notwendigen Eigenschaften sind in der Datenabhängigkeitsanalyse abgebildet. Bei einer Erweiterung des Systems, die mehrere Zuweisungen an eine einzelne Pointer-Variable erlaubt, müssen jedoch neue Zelltypen eingeführt werden, die Zuweisungen an dereferenzierte Pointer darstellen.

4.3 Andere auf Zellularautomaten übertragene Konzepte

Abgesehen von Pointern und Arrays umfasst die C-Programmierung weitere Konstrukte und Verfahren, die die Flexibilität des Systems in Bezug auf die damit zu kompilierenden und auszuführenden Algorithmen erhöhen. Dazu gehören Funktionen mit Eingabeparametern, die gleich ablaufende Berechnungen mit unterschiedlichen Werten ermöglichen, die der Benutzer vorgibt. Des Weiteren wird die Verwendung von Fließkommazahlen in Berechnungen sowie die Übersetzung von globalen Variablen und Funktionen der C-Standard-

Bibliothek (am Beispiel der Funktion `printf`) auf Zellularautomaten in diesem Kapitel erläutert.

4.3.1 Funktionen mit Eingabeparametern

Neben der Kompilierung vollständiger Programme, ausgehend von einer *main*-Funktion, können auch einzelne Funktionen auf einen Zellularautomaten abgebildet werden. Die Auswahl erfolgt durch den Benutzer des Frameworks mittels Markierung des zu kompilierenden Knotens im Syntaxbaum des Projektes (siehe Kapitel 3.2.1).

Ist die Funktion kompiliert und in einen Zellularautomaten übertragen, kann der Benutzer die zur Ausführung benötigten Eingabeparameter festlegen und die Auswertung des Zellularautomaten auf der gewünschten Architektur starten. Beim Start werden die Eingabeparameter durch die Laufzeitumgebung in dafür vorgesehene *Eingabezellen* übertragen. Sie entsprechen in ihrer Funktion den in Kapitel 3.5.1 beschriebenen Wertezellen. Ihr (während der Laufzeit des Zellularautomaten) konstanter Wert wird jedoch erst beim Start der Auswertung festgelegt.

Wird die CPU zur Auswertung des generierten Zellularautomaten verwendet, können die Parameter direkt an die entsprechenden Zellen weitergegeben werden, da sich die Speicherbereiche des Compilers und der virtuellen Maschine nicht unterscheiden. Zur Ausführung auf der GPU werden die Parameter in einen dafür im Arbeitsspeicher der Grafikkarte reservierten Speicherbereich übertragen. Anschließend wird ein Start-Kernel aufgerufen, der die Pointer innerhalb der Eingabezellen auf die dazugehörigen Speicherpositionen der Parameter setzt.

Wählt der Benutzer einen FPGA als Verarbeitungsarchitektur, müssen ähnlich zur GPU-basierten Auswertung die Parameter an den FPGA übertragen werden. Dazu wird in der Software ein Array angelegt, in das alle Werte eingetragen werden. Beim Start der Ausführung wird das Array über die SIRC-Schnittstelle (siehe Kapitel 3.7.3) an den FPGA übermittelt. Der auf dem FPGA implementierte Zustandsautomat, der den Zellularautomaten steuert und überwacht, liest die Parameter vor dem Start der Hauptzelle aus dem Puffer der SIRC-Schnittstelle und schreibt sie in dafür vorgesehene Signalleitungen, die mit den Eingabezellen verbunden sind.

4.3.2 Implementierung weiterer Datentypen

Die in Kapitel 3 beschriebenen Zellen beinhalten einen ganzzahligen Datentyp, der für die Repräsentation ihres Wertes genutzt wird. In der Softwareimplementierung für die CPU und GPU handelt es sich dabei um einen 32-Bit Integer-Datentyp. Für die Hardwareimplementierung wird für den regulären Datenaustausch der Zellen ebenfalls ein 32-Bit breiter Integer-Wert verwendet. Zur Übertragung von Bedingungen für *if*-Zellen oder *while*-Zellen wird hingegen nur ein einzelnes Bit verwendet (siehe Kapitel 3.7.2).

Während ganzzahlige Werte für viele Problemstellungen ausreichend sind, benötigen mathematische Algorithmen oftmals Darstellungen von reellen Zahlen, um ein Problem durch Berechnungen zu lösen. Eine approximative Repräsentation reeller Zahlen bieten so genannte Fließkommazahlen [37]. Bei Fließkommazahlen wird eine Zahl in Exponentialschreibweise durch eine Mantisse und den dazugehörigen Exponenten dargestellt. Die Kommaposition innerhalb der Zahl ist durch die Berechnung variabel und löst das Problem des vor Ausführung eines Algorithmus zu berechnenden Wertebereichs der Vor- und Nachkommastellen von Festkommazahlen. Je mehr Bits für die Speicherung von Mantisse und Exponent verwendet werden, desto genauer lässt sich eine darzustellende Zahl approximieren.

Für die vorliegende Arbeit wurde zusätzlich zum Integer-Datentyp der in C/C++ existierende `float`-Datentyp zur Darstellung von Fließkommazahlen mit einfacher Genauigkeit (32-Bit) in den Zellen des Zellularautomaten implementiert. Bei diesem Datentyp besteht die Mantisse aus 8 Bit, der Exponent umfasst 23 Bit und 1 Bit wird zur Speicherung des Vorzeichens der Zahl verwendet [51]. Eine Erweiterung um Fließkommazahlen mit doppelter Genauigkeit (`double`, 64-Bit) sowie Implementierung anderer C-Datentypen, zum Beispiel `char`, `short int` oder `bool`, kann für Weiterentwicklungen nach dem im folgenden Abschnitt erläuterten Prinzip erfolgen.

In der Softwareimplementierung werden für die Erweiterung die Pointer auf den Integer-Wert der Zellen entfernt und durch Pointer auf eine Basisklasse, die einen abstrakten Datentyp repräsentiert, ersetzt. Die Klasse enthält einen Parameter (gespeichert in einem Byte), der angibt, welchen Datentyp sie darstellt sowie Konvertierungsmethoden zum Umwandeln der Datentypen und Methoden zur Durchführung von Berechnungen (beispielsweise +, -, ·, /).

Die Implementierung der Methoden innerhalb der Basisklasse ist notwendig, um Berechnungen durchführen zu können, bei denen die Operanden unterschiedliche Datentypen haben und somit der Datentyp des Ergebnisses variieren kann. Die in C gültigen Regeln zur Bildung des Ergebnisdatentyps behalten ihre Gültigkeit (zum Beispiel: „`int`“ + „`float`“ = „`float`“).

Aus der Basisklasse abgeleitete Klassen beinhalten zusätzlich eine Variable, die den tatsächlichen Wert in dem korrekten Datentyp speichert. Der Nachteil des angewandten Verfahrens besteht in der Vergrößerung des zu speichernden Wertes einer Zelle um das Byte, in dem der Datentyp der Zelle hinterlegt wird. Die Verwendung einer abstrakten Basisklasse und beliebig daraus ableitbarer Datentypen bietet jedoch mehr Flexibilität in Bezug auf die damit zu kompilierenden Algorithmen.

Eine Umsetzung und Nutzung der unterschiedlichen Datentypen in VHDL und Abbildung auf einen FPGA ist ebenfalls möglich, erfordert jedoch zusätzliche Maßnahmen des Codegenerators und neue Zellmodule. Im Gegensatz zu Integer-Werten und Operationen zwischen ihnen werden Fließkommazahlen vom verwendeten Synthesetool nicht direkt unterstützt. Um die Möglichkeit zur Verarbeitung von Fließkommazahlen herzustellen, wird für das in der vorliegenden Arbeit beschriebene System eine *Floating-Point-Unit* (FPU) in Form eines IP-Cores von *OpenCores* [1] eingesetzt. Eine Operatorzelle innerhalb des Zellularautomaten, die eine Fließkommaberechnung ausführen soll, nutzt die von der FPU bereitgestellten Funktionen. Nach Abschluss der Berechnung setzt die Zelle ihr eigenes Ergebnis auf das der FPU.

Durch den bitgenauen Zugriff und die Definition der Zellnachbarschaften während der Codegenerierung ist in der Hardwareimplementierung keine zusätzliche Speicherung des Datentyps notwendig. Werden neue Datentypen für Weiterentwicklungen des Systems hinzugefügt, so muss der Codegenerator entsprechend angepasst werden.

Für die Übertragung der Fließkommazahlen vom PC zum FPGA und umgekehrt werden die Werte, wie in Kapitel 3.7.3 bzw. 4.3.1 beschrieben, byteweise in ein Array geschrieben und über die SIRC-Schnittstelle versendet. Eine gesonderte Umwandlung ist nicht notwendig, da sowohl der eingesetzte Softwarecompiler (*Microsoft Visual Studio 2010* [75]) als auch die verwendete FPU die *IEEE-Norm 754* [51] einhalten und somit die gleiche interne Datenrepräsentation verwenden.

4.3.3 Globale Variablen

Globale Variablen werden in C-Programmen für den funktionsübergreifenden Datenaustausch verwendet. Wird ein Wert einer globalen Variable durch eine Funktion verändert, so ist der neue Wert für alle anderen Funktionen ebenfalls gültig. Eine rein lokale Datenabhängigkeitsanalyse innerhalb von Funktionen ist zur korrekten Verarbeitung nicht ausreichend. Die Schreib- und Lesezugriffe auf globale Variablen müssen funktionsübergreifend erkannt und die Datenabhängigkeiten für die parallele Ausführung in ϕ-Funktionen gespeichert werden.

Die eingesetzte Methode zur Erkennung von globalen Datenabhängigkeiten basiert auf der in Kapitel 4.2 beschriebenen Implementierung von Pointern. Anstatt der Einführung eines zusätzlichen Verfahrens zur Behandlung globaler Variablen werden sie in lokale Variablen mit Zugriffen über Pointer umgewandelt. Diese Aufgabe übernimmt der Präcompiler (siehe Kapitel 3.3). Er verschiebt außerhalb von Funktionen definierte globale Variablen in die Hauptfunktion (*main*-Funktion oder durch den Benutzer ausgewählte Funktion) des Algorithmus.

Jeder Funktion, die Lese- bzw. Schreibzugriffe auf eine globale Variable beinhaltet, wird die Adresse der jeweiligen Variable als Parameter übergeben. Die Aufgabe des Präcompilers besteht darin, die Funktionsparameter um die notwendigen Pointer zu erweitern und die Aufrufe der Funktionen entsprechend anzupassen. Weiterhin ersetzt der Präcompiler die Zugriffe auf die globalen Variablen innerhalb der Funktionen durch Zugriffe auf die entsprechenden dereferenzierten Pointer und somit auf den im Speicherbereich der Variable hinterlegten Wert.

Zusätzliche Implementierungen von Zelltypen oder spezielle Anpassungen des Compilers sind durch die Änderungen des Präcompilers am zu kompilierenden Algorithmus nicht notwendig. Die Datenabhängigkeitsanalyse erkennt die Beziehungen der Schreib- und Lesezugriffe durch die Verbindung der Funktionen über Pointer.

Abbildung 4.7 zeigt die durch den Präcompiler durchgeführten Änderungen zur Überstzung globaler Variablen an einem Codebeispiel.

```
int x = 0;

int funcA()
{
  x = x >> 2;

  return -x;
}

void funcB()
{
  int y = funcA();

  x = ( y + 2 ) + x;
}

int get10( int a )
{
  x = a;

  funcB();

  return x;
}
```

(a) Originaler C-Code mit globaler Variable

```
int funcA( int* x_p )
{
  *x_p = *x_p >> 2;

  return -(*x_p);
}

void funcB( int* x_p )
{
  int y = funcA( x_p );

  *x_p = ( y + 2 ) + *x_p;
}

int get10( int a )
{
  int x = a;

  funcB( &x );

  return x;
}
```

(b) Veränderter Code mit Pointern

Abbildung 4.7: Ersetzung globaler Variablen durch lokale Variablen und Zugriffe über Pointer

4.3.4 Funktionen der C-Standard-Bibliothek am Beispiel der Funktion „printf“

Die Funktionen der C-Standard-Bibliothek sind ein elementarer Bestandteil der C-Programmiersprache und seit C89 [5] bzw. C90 [59] durch internationale Standards ratifiziert. Sie bieten beispielsweise Möglichkeiten zur Eingabe von Werten über die Konsole und Ausgabe von Daten an den Benutzer durch Darstellung auf dem Bildschirm.

Die Ausgabe auf dem Bildschirm wird in C durch die Funktion `printf` realisiert. Sie übernimmt einen String, der die Formatierung der Ausgabe beschreibt, und die darzustellenden Werte. Zusätzlich zur Ausgabe der finalen Berechnungswerte eines Algorithmus kann sie dazu verwendet werden, den Benutzer während der Laufzeit von Programmen über den aktuellen Status zu informieren. Des Weiteren ist die Funktion hilfreich für die Entwicklung von

Algorithmen und zur Überprüfung des Ablaufes durch Zwischenausgaben an den Programmierer.

Da C-Algorithmen von Programmierern in sequentieller Abfolge geschrieben werden und sie eine dementsprechende sequentielle Ausgabe der `printf`-Funktion erwarten, muss diese bei der Übertragung in das Modell der Zellularautomaten während der Datenabhängigkeitsanalyse berücksichtigt werden. Die Behandlung der `printf`-Funktion erfolgt dabei analog zu dem auf globale Variablen angewendeten Konzept (siehe Kapitel 4.3.3). Der Präcompiler ersetzt alle Aufrufe der `printf`-Funktion durch Pointerzugriffe, die in der Datenabhängigkeitsanalyse zu einer Serialisierung führen. Jeder Aufruf der `printf`-Funktion ist vom algorithmisch vorhergehenden Aufruf abhängig. Dadurch wird gewährleistet, dass die Ausgabe der vom Programmierer bzw. Benutzer erwarteten Reihenfolge entspricht.

Aus Sicht der Datenabhängigkeitsanalyse entspricht ein Aufruf der `printf`-Funktion einer bzw. mehreren (bei mehr als einem Argument) `+=`-Zuweisung/en an die globale `printf`-Variable. Zur Umsetzung der Funktionalität in den Zellularautomaten muss ein neuer Zelltyp hinzugefügt werden, der den `printf`-Aufruf durchführt. Eine `printf`-Zelle erhält bei ihrer Erzeugung durch den Compiler den einzusetzenden Formatierungsstring sowie die zu verwendenden Argumente als konstante Werte bzw. ϕ-Funktionen. Während der softwarebasierten Auswertung der Zellularautomaten führt eine Aktivierung der `printf`-Zelle bei Vorliegen der einzusetzenden Argumente zu einem Aufruf der in der virtuellen Maschine über die C-Standard-Bibliothek vorhandenen `printf`-Funktion. Der Formatierungsstring und die zu verwendenden Argumente werden dabei entsprechend eingesetzt.

Eine Implementierung der `printf`-Zellen für die hardwarebasierten Zellularautomaten ist momentan nicht vorgesehen, ist jedoch über einen zusätzlichen Speicherbereich auf dem FPGA realisierbar. Der Speicherbereich auf dem FPGA kann von allen `printf`-Zellen als Puffer benutzt werden, der entweder am Ende der Auswertung des Algorithmus zusammen mit den Rückgabewerten übertragen oder bereits während der Auswertung durch die *SIRC*-Schnittstelle (siehe Kapitel 3.7.3) auslesbar ist. Nach Erhalt der Formatierungsdaten und Argumente muss die Steuerungssoftware auf dem PC die Daten für den Benutzer anzeigen bzw. ausgeben.

5 Experimentelle Untersuchung verschiedener Algorithmen/Zellularautomaten

Gegenstand des folgenden Kapitels sind experimentelle Untersuchungen der in Kapitel 3 und Kapitel 4 beschriebenen Verfahren und Technologien. Diese wurden auf unterschiedliche Algorithmen angewendet und auf ihre Praxiseignung untersucht. Es werden Vergleiche zwischen dem softwarebasierten und hardwarebasierten Ansatz vorgenommen sowie Unterschiede zur Übersetzung von C-Programmen durch Standardcompiler aufgezeigt.

Alle Experimente wurden auf einem PC mit Intel Core i7-2600K Prozessor [55] mit 4 Rechenkernen sowie Hyper-Threading-Technologie und somit 8 parallel ausführbaren Threads durchgeführt. Des Weiteren verfügt der für die Experimente verwendete PC über eine NVIDIA GeForce 580 GTX Grafikkarte [80] mit 512 Rechenkernen, die für die Ausführung der GPU-basierten virtuellen Maschine eingesetzt wurde. Für die Auswertung der Zellularautomaten in Hardware wurde ein Xilinx XUPV5-LX110T Entwicklungsboard [116] mit einem Virtex-5 XC5VLX110T FPGA verwendet. Der durch den Codegenerator (siehe Kapitel 3.7.1) erzeugte VHDL-Code wurde durch die *Xilinx ISE Design Suite* [115] synthetisiert und auf den FPGA abgebildet.

5.1 Untersuchung des Laufzeitverhaltens verschiedener Zellularautomaten

Die in diesem Kapitel vorgestellten Untersuchungen beschränken sich auf Algorithmen ohne Arrays und Pointer, die durch die in Kapitel 3 beschriebenen Zellularautomaten (inklusive der Erweiterung um variable Eingabeparameter aus Kapitel 4.3.1) verarbeitet werden können.

Anhand der Algorithmen werden verschiedene Eigenschaften der Zellularautomaten untersucht, die parallele Zellauswertung überprüft, die Auswirkungen von Loop-Unrolling getestet sowie Vergleiche zwischen hardware- und softwarebasierter Auswertung angestellt.

```c
int func( int x, int y )
{
  return x * y;
}

int main()
{
  int x, res1;

  for ( x = 0; x < 100; x++ )
    res1 += func( x, x );

  return res1;
}
```

(a) exp0_a.c

```c
int func( int x, int y )
{
  return x * y;
}

int main()
{
  int x, y, res1, res2;

  for ( x = 0; x < 100; x++ )
    res1 += func( x, x );
  for ( y = 0; y < 100; y++ )
    res2 += func( y, y );

  return res1 + res2;
}
```

(b) exp0_b.c

```c
int func( int x, int y )
{
  return x * y;
}

int main()
{
  int x, res1, res2;

  for ( x = 0; x < 100; x++ )
    res1 += func( x, x );
  for ( x = 0; x < 100; x++ )
    res2 += func( x, x );

  return res1 + res2;
}
```

(c) exp0_c.c

Abbildung 5.1: C-Code für Experiment 1

5.1.1 Experiment 1 - Parallele Ausführung voneinander unabhängiger Schleifen

In einem ersten Experiment wurde die automatische parallele Ausführung voneinander unabhängiger Schleifen durch den generierten Zellularautomaten untersucht. Abbildung 5.1(a-c) enthält die dazu verwendeten Algorithmen.

Das Beispiel in Abbildung 5.1(a) beinhaltet eine Schleife, deren Ausführung von einer Zählvariable (`x`) abhängt. Der zweite Algorithmus (Abbildung 5.1(b)) stellt eine Erweiterung des ersten Algorithmus um eine zusätzliche Schleife dar,

Name	exp0_a.c	exp0_b.c	exp0_c.c
Zellen	34	68	66
Generationen	1610	1611	1611
Laufzeit in ms	2,008	2,253	2,247

Tabelle 5.1: Größe, benötigte Zellgenerationen und Gesamtlaufzeiten der Zellularautomaten in Experiment 1

deren Abarbeitung autonom von der ersten Schleife ist. Die Unabhängigkeit der Schleifen wird durch den Einsatz verschiedener Zählvariablen (`x` für die erste Schleife und `y` für die zweite Schleife) erreicht. Da zwischen den Variablen keine Datenabhängigkeiten bestehen, können die Schleifen von dem, aus dem Algorithmus generierten, Zellularautomaten parallel verarbeitet werden.

Die parallele Ausführung spiegelt sich in der Ausführungszeit der Algorithmen bis zum Abschluss ihrer Berechnungen wieder. Tabelle 5.1 stellt die Anzahl an benötigten Zellupdates bis zur vollständigen Auswertung der Algorithmen (Generationen), die damit verbundenen Laufzeiten sowie die Anzahl an erzeugten Zellen gegenüber. Zur experimentellen Auswertung der Zellularautomaten wurde die CPU-basierte virtuelle Maschine (siehe Kapitel 3.6.4) verwendet. Die Laufzeiten stellen zur Reduzierung von Messtoleranzen einen Mittelwert über jeweils 100.000 Durchläufe des Experiments dar. Die Zellgeneration wurden im Debug-Modus der virtuellen Maschine ermittelt, um einen (über mehrere Durchläufe der Experimente) konstanten Verlauf unabhängig von asynchroner Zellkommunikation zu erhalten.

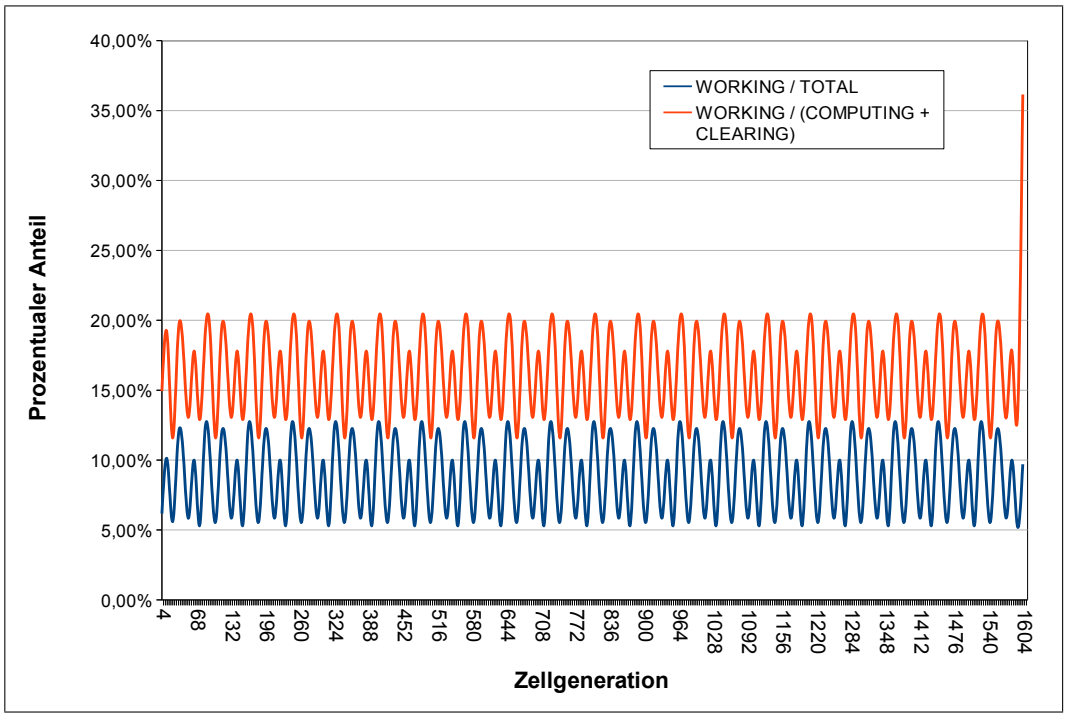
Die Übersetzung des Algorithmus *exp0_b.c* führt, verglichen mit *exp0_a.c*, zu einer doppelt so hohen Anzahl an Zellen im kompilierten Zellularautomaten. Seine Laufzeit ist hingegen lediglich 12% länger. Betrachtet man die Anzahl an benötigten Zellupdates bis zur vollständigen Auswertung der Zellularautomaten, so unterscheiden sie sich um eine Generation. Diese zusätzliche Zellgeneration im zweiten Algorithmus entsteht durch die Bildung der Summen der Einzelergebnisse `res1` und `res2` in Codezeile 15. Die Differenz des Overheads zwischen der Anzahl an Zellgenerationen und tatsächlicher Laufzeit der Algorithmen ist auf den zusätzlichen Verwaltungsaufwand der Zellen von Algorithmus *exp0_b.c* durch die CPU zurückzuführen. Während bei Algorithmus *exp0_a.c* jeder (virtuelle) Prozessorkern des für das Experiment eingesetzten

Systems circa 4 (4,25) Zellen verwaltetet, sind es bei Algorithmus *exp0_b.c* circa 8 (8,5) Zellen. Die Prozessorkerne müssen für Algorithmus *exp0_b.c* pro Update des Zellularautomaten doppelt so viele Zellen, verglichen mit der Zellanzahl von Algorithmus *exp0_a.c*, bearbeiten. Das führt zu einer Verlängerung der Bearbeitungszeit der einzelnen Updates und somit zu einer Verlängerung der Gesamtlaufzeit.

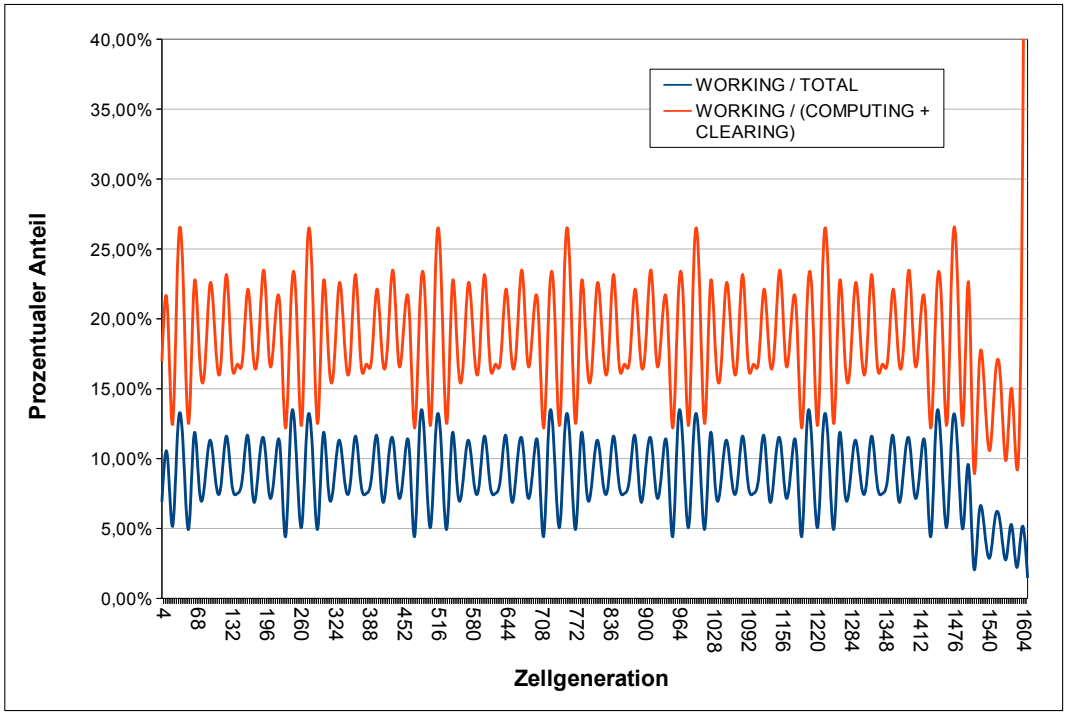
Abbildung 5.2 zeigt die Auslastung der Zellen während der Auswertung der Algorithmen. Die Kurve *WORKING/TOTAL* stellt das Verhältnis der Anzahl an *arbeitenden* Zellen im Bezug zur Gesamtzahl an Zellen des jeweiligen Zellularautomaten dar. Arbeitende Zellen sind all jene Zellen, die in der jeweils aktuellen Generation des Zellularautomaten eine Aktion ausgeführt haben, die ihren Ausgabezustand verändert hat. Das beinhaltet sowohl den Abschluss ihrer Auswertung (Wechsel von *COMPUTING* auf *EVAL_TRUE* oder *EVAL_FALSE*) als auch den Abschluss des Zurücksetzens ihrer Werte (Wechsel von *EVAL_TRUE* oder *EVAL_FALSE* auf *UNDEFINED*). Die Kurve *WORKING/TOTAL* beschreibt somit die Auslastung des gesamten Zellularautomaten.

Das Verhältnis von arbeitenden Zellen zu der Anzahl an Zellen, deren Eingabezustand auf *COMPUTING* bzw. *CLEARING* gesetzt ist, stellt die Kurve *WORKING/(COMPUTING + CLEARING)* dar. Sie beschreibt die Beziehung zwischen auswertbaren und wartenden Zellen pro Zellgeneration. Beispielsweise bedeutet eine Auslastung von 10% für eine Zellgeneration, dass 10% der Zellen deren aktuelle Aufgabe gestartet wurde, ihre Auswertung in der betrachteten Zellgeneration abschließen konnten. Die restlichen 90% sind von den Ergebnissen anderer Zellen abhängig und müssen auf diese warten, bevor sie in den folgenden Generationen ihre Aufgabe beenden können.

Beide Algorithmen (*exp0_a.c* und *exp0_b.c*) zeigen eine ähnliche Auslastung (siehe Abbildung 5.2(a) bzw. 5.2(b)). Die Mittelwerte der totalen Auslastung über den gesamten Auswertungszeitraum liegen bei circa 8,79% für beide Algorithmen. Das Verhältnis von tatsächlich arbeitenden zu gestarteten Zellen liegt im Mittel bei 16,29% für Algorithmus *exp0_a.c* und 18,64% für Algorithmus *exp0_b.c*. Trotz der doppelt so hohen Anzahl an Zellen im zweiten Zellularautomaten ist die Auslastung nahezu identisch, was gleichbedeutend mit einem höheren Parallelitätsgrad des Algorithmus ist. Die parallele Ausführung der



(a) exp0_a.c



(b) exp0_b.c

Abbildung 5.2: Prozentualer Anteil an arbeitenden Zellen für jede Zellgeneration bei der Auswertung der Algorithmen exp0_a.c und exp0_b.c

voneinander unabhängigen Schleifen ist der Grund für die geringe Differenz der Anzahl an benötigten Zellgenerationen bis zur vollständigen Bearbeitung der Algorithmen. Das führt dazu, dass die Gesamtlaufzeit von Algorithmus *exp0_b.c* lediglich 12% über der von *exp0_a.c* liegt, obwohl doppelt so viele Berechnungen innerhalb des Algorithmus durchgeführt werden müssen.

Algorithmus *exp0_c.c* (siehe Abbildung 5.1(c)) ist eine veränderte Version von Algorithmus *exp0_b.c*. Anstatt einer zusätzlichen Zählvariable für die zweite Schleife wird die Variable `x` als Zählvariable beider Schleifen eingesetzt. Während sich die Größe des kompilierten Zellularautomaten um die zwei Zellen reduziert, die für die Speicherung und Initialisierung von `y` in Algorithmus *exp0_b.c* notwendig waren, ändert sich weder die Anzahl an benötigten Zellupdates zur Auswertung noch wird die Laufzeit des Algorithmus negativ beeinflusst (siehe Tabelle 5.1).

Dieser Effekt entsteht durch die automatisch ermittelten Datenabhängigkeiten und die interne SSA-Darstellung des Compilers (siehe Kapitel 3.4.1). Auch wenn der Benutzer (Programmierer) die Zählvariablen nicht explizit voneinander trennt, so erkennt der Compiler die Unabhängigkeit der Instanzen der Variablen und führt die Schleifen dennoch parallel aus.

5.1.2 Experiment 2 - Effizienzsteigerung durch Loop-Unrolling

Wie in Kapitel 3.3.2 beschrieben, besteht durch Loop-Unrolling eine Möglichkeit, die Effizienz eines Programms, insbesondere bei paralleler Ausführung, zu steigern. Die Auswirkungen von Loop-Unrolling auf die Laufzeit und das Verhalten eines Zellularautomaten wurden experimentell untersucht und die Ergebnisse werden im Folgenden dargestellt.

Abbildung 5.3 stellt den für dieses Experiment verwendeten Algorithmus mit zwei ineinander verschachtelten Schleifen und den zugehörigen Zählvariablen `x` und `y` dar. Beide Variablen nehmen während der Ausführung die Werte 0 bis 9 an, so dass die Funktion `func` insgesamt 100-mal aufgerufen und ihr Ergebnis aufsummiert wird.

Loop-Unrolling kann mit verschiedenen Strategien angewendet werden, die zu unterschiedlichen Ergebnissen in Bezug auf die Größe des aus dem Algorithmus generierten Zellularautomaten führt. Weiterhin hat die Wahl der

```
1  int func( int x, int y )
2  {
3    return x + y;
4  }
5
6  int main()
7  {
8    int x, y;
9    int res = 0;
10
11   for ( x = 0; x < 10; x++ )
12     for ( y = 0; y < 10; y++ )
13       res += func( x, y );
14
15   return res;
16 }
```

Abbildung 5.3: C-Code für Experiment 2

Name	Original	LU→x	LU→x,y	LU'→x	LU'→x,y
Zellen	59	288	1030	324	1064
Gen_{CPU}	1850	1541	212	189	43
Gen_{GPU}	2293	2048	223	246	52
Zeit_{CPU}	2,408 ms	2,814 ms	0,548 ms	0,416 ms	0,144 ms
Zeit_{GPU}	23,358 ms	24,888 ms	2,184 ms	3,318 ms	0,799 ms

Tabelle 5.2: Größe, benötigte Zellgenerationen (Gen) und Gesamtlaufzeiten (Zeit) der Zellularautomaten in Experiment 2

Loop-Unrolling-Strategie Einfluss auf die Anzahl an benötigten Zellgenerationen zur Auswertung des jeweiligen Algorithmus und auf die Gesamtlaufzeit des kompilierten Programms.

Tabelle 5.2 enthält die experimentell ermittelten Größen der Zellularautomaten, die Anzahl an benötigten Zellgenerationen sowie die Laufzeiten zur Auswertung der Algorithmen. Zur Reduzierung von Messtoleranzen stellen die Zellgenerationen und Laufzeiten Mittelwerte über 100.000 Durchläufe des Experiments dar.

Die erste für dieses Experiment genutzte Strategie (LU → x) wendet Loop-Unrolling auf die äußere (von `x` abhängige) Schleife des Beispielalgorithmus an. Die Schleife wird durch den Präcompiler (siehe Kapitel 3.3) vollständig aufgelöst, die inneren Schleifen (mit der Zählvariable `y`) werden 10-mal hin-

tereinander kopiert und die entsprechenden Werte für `x` beim Funktionsaufruf von `func` eingesetzt. Der positive Effekt dieser Strategie bleibt jedoch aus, wie Tabelle 5.2 zu entnehmen ist. Zwar reduziert sich die Anzahl an insgesamt benötigter Zellupdates, da ein Zurücksetzen der äußeren Schleife entfällt, die Laufzeiten für die auf der CPU und GPU ausgeführten Zellularautomaten werden jedoch länger (verglichen mit der Laufzeit des originalen Algorithmus).

Dieser negative Effekt entsteht durch die Datenabhängigkeiten der Variable `res`. Die Erhöhung des Wertes durch den `+=`-Operator impliziert ein Lesen des vor der Operation gültigen Wertes. Die Reihenfolge der Abarbeitung ist somit strikt vorgegeben und führt dazu, dass die Auswertung der Schleifen voneinander abhängig ist und sequentiell durchgeführt werden muss. Die zweite Schleife kann ihre erste Iteration erst abschließen (den `+=`-Operator für `res` anwenden) wenn die erste Schleife vollständig abgearbeitet ist. Auf die gleiche Art und Weise müssen alle folgenden Schleifen innerhalb des Zellularautomaten behandelt werden.

Verdeutlicht wird das Verhalten des Zellularautomaten bei Betrachtung von Abbildung 5.4. Das Diagramm beinhaltet die verschiedenen Loop-Unrolling-Strategien und das Verhalten der daraus resultierenden Zellularautomaten. Die x-Achse des Diagramms beschreibt den prozentualen Fortschritt der Auswertung im Bezug auf die Gesamtlaufzeit der jeweiligen Algorithmen. Die y-Achse stellt den prozentualen Anteil an ausgewerteten (Zwischen-)Ergebnissen im Bezug zur für jeden Zellularautomaten individuellen Gesamtzahl an (Zwischen-)Ergebnissen dar.

Der durch die oben beschriebene erste Strategie (LU $\rightarrow$ x) erzeugte Zellularautomat zeigt lediglich in seiner Anfangsphase ein vom originalen Zellularautomaten differenziertes Verhalten. In dieser Phase werden die zehn Schleifen parallel initialisiert und bis an die Stelle des `+=`-Operators ihrer ersten Iteration ausgewertet. Die parallele Verarbeitung bis zu der genannten Stelle ist durch eine steil ansteigende Kurve im Diagramm erkennbar. Anschließend flacht die Kurve ab und steigt mit geringerem Wachstum als die Kurve, die das Verhalten des originalen Algorithmus wiedergibt.

Die zweite Strategie (LU $\rightarrow$ x, y) veranlasst den Präcompiler neben der äußeren Schleife mit der Zählvariable `x` auch die innere Schleife mit der Zählvariable `y` aufzulösen. Die 10 durch Strategie (LU $\rightarrow$ x) erzeugten Schleifen

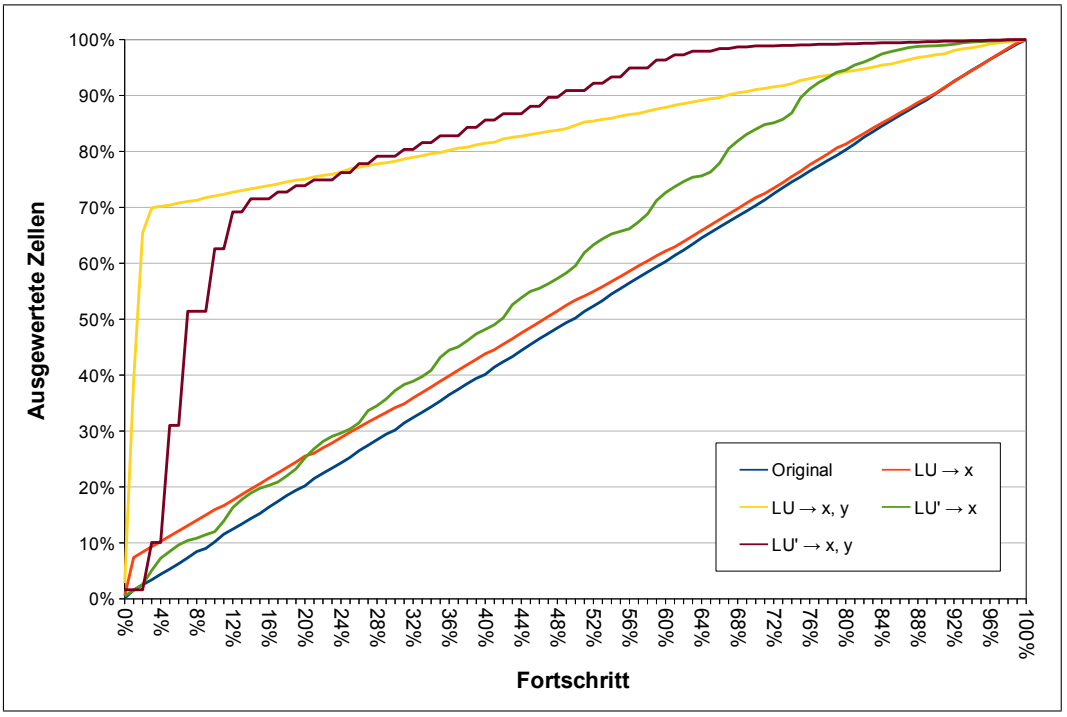


Abbildung 5.4: Vergleich verschiedener Loop-Unrolling-Strategien und ihr Effekt auf das Verhalten des Zellularautomaten

werden durch jeweils 10 Kopien der in der Schleife über `y` enthaltenen Anweisungen ersetzt und die entsprechenden Werte für `y` beim Aufruf der Funktion `func` eingesetzt.

Der durch diese Strategie veränderte Algorithmus und der daraus generierte Zellularautomat beinhaltet circa 17-mal so viele Zellen, verglichen mit dem originalen Zellularautomaten. Die dadurch ermöglichte parallele Ausführung führt dazu, dass der Zellularautomat nur circa 11% der ursprünglichen Zellupdates benötigt, um den Algorithmus vollständig abzuarbeiten. Die Anzahl an Zellgenerationen und die unabhängige Verarbeitung führen außerdem zu einer Beschleunigung der Auswertung um Faktor 4 (4,39) für die CPU bzw. Faktor 10 (10,69) für die Ausführung des Zellularautomaten auf der GPU (siehe Tabelle 5.2).

Die Gemeinsamkeit zur Strategie (LU → x) besteht in den Datenabhängigkeiten der Zugriffe auf die Instanzen der Variable `res` durch den `+=`-Operator. Da im Gegensatz zu Strategie (LU → x) für den Zellularautomaten keine Schleifendurchläufe mit voneinander abhängigen Iterationen zu absolvieren sind, können die übrigen Ergebnisse (Auswertung der Funktion `func` für alle `x`

und y) bereits vorab parallel berechnet werden. Anschließend wird durch den Zellularautomat sequentiell die Summe aller Zwischenergebnisse gebildet.

Das beschriebene Verhalten dieses Zellularautomaten spiegelt sich im Diagramm in Abbildung 5.4 wieder. Mehr als 70% der auszuwertenden Zellen/-Ergebnisse liegen bereits nach weniger als 5% der Gesamtlaufzeit des Zellularautomaten vor, was sich im Diagramm durch eine steil wachsende Kurve bemerkbar macht. Anschließend (ab 5% der Gesamtlaufzeit) flacht die Kurve ab und steigt mit geringerem Wachstum als die Kurve des originalen und durch Strategie (LU $\rightarrow$ x) erzeugten Zellularautomaten.

Strategie (LU' $\rightarrow$ x) ist eine Abwandlung der bereits vorgestellten Strategie (LU $\rightarrow$ x). Zusätzlich zum Loop-Unrolling der äußeren Schleife wurden hierbei die Datenabhängigkeiten der Schleifen untereinander aufgelöst. Dazu wurden lokale Variablen zur Speicherung der Zwischenergebnisse jeder Schleife in den Algorithmus eingefügt. Am Ende des Algorithmus wird die Gesamtsumme durch Aufsummierung der 10 Zwischenergebnisse erzeugt. Diese Strategie kann in der momentanen Version des Frameworks nicht automatisch vom Präcompiler vorgenommen werden und wurde für das Experiment manuell durchgeführt. Auflösung von Datenabhängigkeiten, insbesondere bei der Anwendung von Loop-Unrolling, bietet eine Optimierungsmöglichkeit für zukünftige Weiterentwicklungen des Systems.

Die parallele Berechnung der Zwischenergebnisse der Schleifen und das anschließende Aufsummieren durch den auf die 10 lokalen Variablen angewendeten +=-Operator zeigt sich im Diagramm in Abbildung 5.4 durch einen wellenförmigen Verlauf der Fortschrittskurve. Die Kurve steigt zu Beginn jeder durch die 10 Schleifen gleichzeitig ausgeführten Schleifeniteration stark an und flacht am Ende der Iteration wieder ab.

Der durch die Strategie (LU' $\rightarrow$ x) erzeugte Zellularautomat vereint die positiven Eigenschaften der Strategien (LU $\rightarrow$ x) und (LU $\rightarrow$ x, y). Die Anzahl an generierten Zellen entspricht annähernd der Anzahl der durch Strategie (LU $\rightarrow$ x) generierten. Die Anzahl an benötigten Zellupdates zur Auswertung des Zellularautomaten und die Gesamtlaufzeit des Algorithmus sind zudem geringer (bezogen auf die CPU basierte Auswertung) als die des durch Strategie (LU $\rightarrow$ x, y) gebildeten Zellularautomaten.

Die Auswertung auf der GPU ist (im Gegensatz zur CPU-basierten Variante) langsamer als die Auswertung des durch Strategie (LU → x, y) erzeugten Zellularautomaten. Dieser Effekt entsteht durch die geringere Anzahl an Zellen und somit niedrigere Auslastung der GPU-Rechenkerne. Ein detaillierter Vergleich zwischen der Ausführung der Zellularautomaten auf der CPU und GPU ist in Kapitel 5.1.3 angegeben.

Die im Bezug auf die Gesamtlaufzeit als auch Anzahl an zur vollständigen Abarbeitung notwendigen Zellgenerationen effizienteste Strategie ist Strategie (LU' → x, y). Hierbei wurde auf die voneinander unabhängigen Schleifen aus Strategie (LU' → x) wiederum Loop-Unrolling angewendet, so dass die Iterationen der Schleifen, bis auf die Aufsummierung der lokalen Variablen, parallel verarbeitet werden können.

Der durch die Strategie (LU' → x, y) erzeugte Zellularautomat benötigt lediglich 2,3% der Zellgenerationen des originalen Zellularautomaten zur Berechnung des Algorithmus. Der Beschleunigungsfaktor, verglichen mit der ursprünglichen Gesamtlaufzeit, liegt für die CPU-Variante bei circa 16,7 und bei 29,2 für die GPU-basierte Auswertung. Der Nachteil dieser Strategie besteht in der Größe des generierten Zellularautomaten. Mit 1064 erzeugten Zellen ist der durch Strategie (LU' → x, y) generierte Zellularautomat circa 18-mal größer als der originale Zellularautomat.

Das Verhalten des durch die Strategie (LU' → x, y) generierten Zellularautomaten entspricht weitestgehend dem Verhalten des durch Strategie (LU → x, y) erzeugten. Im Diagramm in Abbildung 5.4 ist ein starker Anstieg der Fortschrittskurve zu Beginn der Verarbeitung zu erkennen, was die parallele Berechnung der Zwischenergebnisse kennzeichnet. Das Abflachen der Kurve bei circa 18% der Gesamtlaufzeit beschreibt die parallele Bildung der Summen der 10 lokalen Variablen. Das starke Abflachen bei circa 68% beruht auf der anschließenden sequentiellen Bildung der Gesamtsumme der lokalen Variablen.

Insgesamt gesehen bietet Loop-Unrolling eine Möglichkeit die Leistung der in Zellularautomaten übersetzten Algorithmen zu steigern. Es sind jedoch weitere Forschungen notwendig, um stets die bestmögliche Strategie auszuwählen und automatisch durch den Compiler bzw. Präcompiler umsetzen zu lassen.

5.1.3 Experiment 3 - Vergleich zwischen CPU-, GPU- und FPGA-Auswertung

Im Folgenden werden die Performance und die praktischen Unterschiede der softwarebasierten und hardwarebasierten Auswertung von Zellularautomaten anhand von vier Beispielen miteinander verglichen.

Die für die Untersuchung eingesetzten Algorithmen basieren auf dem in [98] vorgestellten Verfahren zur binären Berechnung des *größten gemeinsamen Teilers* zweier Zahlen (`gcd`) sowie dem in [39] beschriebenen Algorithmus zur Berechnung der ganzzahligen Wurzel einer natürlichen Zahl (`sqrt`).

Die Algorithmen wurden ohne *main*-Funktion übersetzt und haben variable Aufrufparameter, deren Werte vor Beginn der Auswertung an die kompilierten Zellularautomaten übermittelt werden (siehe Kapitel 4.3.1). Dadurch wird vermieden, dass die Software zur Übersetzung des erzeugten VHDL-Codes in Hardware und Abbildung auf den FPGA den Algorithmus bereits im Voraus anhand der konstanten Parameter auswertet und nur das Ergebnis des Algorithmus auf den FPGA abbildet. Die Variabilität der Eingabedaten ermöglicht somit eine Vergleichbarkeit der Ergebnisse bezüglich der Laufzeit und Anzahl an Zellgenerationen zwischen Software und Hardware.

Die nachfolgende Auflistung gibt einen Überblick über die untersuchten Algorithmen und die zur Ausführung eingesetzten Aufrufparameter.

Übersetzte Algorithmen und verwendete Aufrufparameter:

1. `gcd( 3528, 3780 )` - Bestimmung des *größten gemeinsamen Teilers* von 3528 und 3780 (Algorithmus nach [98])
2. `sumEuler( 100 )` - Berechnung der Summe der *Eulerschen Phi-Funktion* (Definition in [44]) aller Zahlen von 1 bis 100 (Algorithmus siehe Abbildung 5.5(a))
3. `sumSqrt( 244140625 )` - 1000-mal (in einer Schleife) hintereinander ausgeführte Bestimmung der Wurzel von 5^{12} und Aufsummierung des Ergebnisses (Algorithmus siehe [39] und Abbildung 5.5(b))
4. `sumSqrtLU( 244140625 )` - identischer Algorithmus zu 3. - die Schleife zur Berechnung der Summe wurde jedoch durch den Präcompiler mittels Loop-Unrolling aufgelöst

```c
int sumEuler( int number )
{
  int x, y;
  int sum = 0;

  for ( x = 1; x <= number; x++ )
  {
    for ( y = 1; y <= x; y++ )
    {
      if ( gcd( x, y ) == 1 )
        sum++;
    }
  }

  return sum;
}
```

(a) sumEuler.c

```c
int sumSqrt( int number )
{
  int x = 0;
  int sum = 0;

  for ( x = 0; x < 1000; x++ )
    sum += sqrt( number );

  return sum;
}
```

(b) sumSqrt.c

Abbildung 5.5: C-Code für Experiment 3

Die Größe der aus den Algorithmen generierten Zellularautomaten sowie die Anzahl an Zellgenerationen und die Gesamtlaufzeiten zur Auswertung der Zellularautomaten sind Tabelle 5.3 zu entnehmen. Die ermittelten Zellgenerationen sowie Gesamtlaufzeiten für die CPU und GPU sind Mittelwerte über 100.000 (für den `gcd`-Algorithmus), 100 (`sumEuler` und `sumSqrt`) bzw. 1.000 (`sumSqrtLU`) Durchläufe des Experiments. Für den FPGA sind es durch Simulation mit den nach *Place & Route* (auf dem spezifischen FPGA) erreichbaren Taktfrequenzen ermittelte Werte.

Die CPU wertet die ersten drei Zellularautomaten (`gcd`, `sumEuler` und `sumSqrt`) mit einem Beschleunigungsfaktor zwischen 8,8 und 9,8 verglichen mit der GPU aus. Die Berechnungen des mittels Loop-Unrolling gebildeten Zellularautomaten aus Algorithmus `sumSqrtLU` sind hingegen auf der GPU circa 3-mal schneller als auf der CPU. Dieser Effekt ist, wie in Kapitel 5.1.2 beschrieben, durch die Größe der Zellularautomaten erklärbar.

Während für die ersten drei Algorithmen die eingesetzte CPU mit 8 Threads insgesamt 97 bis 174 Zellen verwaltet, sind auf der GPU bei der Verarbeitung der Algorithmen mehr als 300 der 512 zur Verfügung stehenden Rechenkerne ungenutzt. Zudem ist die GPU mit 772 MHz langsamer getaktet als die CPU mit 3,4 GHz. Für den aus Algorithmus `sumSqrtLU` erzeugten Zellularautoma-

Name	gcd	sumEuler	sumSqrt	sumSqrtLU
Zellen	115	174	97	74.009
Gen_{CPU}	212	1.028.732	400.482	2.370
Gen_{GPU}	252	1.117.822	406.038	2.500
Gen_{FPGA}	54	272.942	85.004	83
Zeit_{CPU}	0,25 ms	1.358,71 ms	453,04 ms	770,42 ms
Zeit_{GPU}	2,46 ms	12.046,61 ms	4.072,61 ms	251,11 ms
$\text{Zeit}_{\text{FPGA}}$	$2,43 \cdot 10^{-4}$ms	1,42 ms	0,38 ms	0,04 ms

Tabelle 5.3: Größe, Anzahl an Zellgenerationen (Gen) und Gesamtlaufzeiten (Zeit) der Zellularautomaten in Experiment 3

ten müssen insgesamt 74.009 Zellen verwaltet werden. In diesem Fall sind pro Zellgeneration von 8 CPU-Threads im Mittel jeweils circa 9251 Zellen zu bearbeiten. Für die GPU-Variante werden 74.009 Threads erstellt, die vom GPU internen Scheduler gesteuert und an die 512 zur Verfügung stehenden Prozessoren zur Auswertung verteilt werden. Jeder Prozessor der GPU muss somit im Mittel circa 144 Zellen pro Generation des Zellularautomaten bearbeiten.

Der theoretische Beschleunigungsfaktor für die GPU (512 Threads mit 772 MHz) verglichen mit der CPU (8 Threads mit 3,4 GHz) bei vollständiger Auslastung beider Systeme liegt bei:

$$\frac{512 \cdot 772}{8 \cdot 3400} \approx 14,53 \ .$$

Der experimentell für das Beispiel ermittelte Faktor von (770,42 ms / 251,11 ms) ≈ 3 weicht jedoch auf Grund des vorhandenen Kommunikationsoverheads sowie der Anzahl an vollständig bearbeitenden Zellen mit steigender Anzahl an Generationen von dem theoretischen Wert ab. Dennoch ist erkennbar, dass eine GPU-basierte Auswertung, bei hinreichend großen Zellularautomaten, einen Performancevorteil gegenüber der CPU-basierten Verarbeitung hat.

Anhand der benötigten Zellgenerationen zum Auswerten der Algorithmen ist ersichtlich, dass die asynchrone Zellkommunikation (siehe Kapitel 3.6.3) im Fall der CPU einen positiven Effekt auf die Gesamtzahl der Updates des jeweiligen Zellularautomaten hat. Da asynchrone Kommunikation lediglich dann genutzt werden kann, wenn die Anzahl an Zellen im Zellularautomaten größer ist als die Anzahl an zur Bearbeitung eingesetzter Threads (siehe Kapitel 3.6.3), kann das

Verfahren von der GPU für die ersten drei Algorithmen nicht genutzt werden. Das resultiert in einer 8,7% bzw. 1,4% höheren Anzahl an Zellgenerationen für Algorithmus `sumEuler` bzw. `sumSqrt` bei der Auswertung durch die GPU verglichen mit der CPU.

Zwar ist eine Nutzung der asynchronen Zellkommunikation durch die GPU für Algorithmus `sumSqrtLU` möglich, die Auswirkung auf Grund der hohen Anzahl an parallelen Threads auf die Gesamtzahl der Zellgenerationen ist jedoch geringer als bei der Auswertung durch die CPU. Das äußert sich im Beispiel durch eine circa 5,5% höhere Anzahl an Zellgenerationen für Algorithmus `sumSqrtLU` bei Bearbeitung durch die GPU verglichen mit der CPU.

Die um 18,9% höhere Gesamtzahl an Updates des Zellularautomaten für die Auswertung des `gcd`-Algorithmus durch die GPU im Vergleich zur CPU resultiert aus der lediglich nach jedem 100. Update durchgeführten expliziten Synchronisation und Überprüfung auf Ergebnisse durch die CPU. Weitere Einzelheiten dazu sind in Kapitel 3.6.4 und im experimentellen Vergleich zwischen Debug- und Release-Modus der virtuellen Maschinen in Kapitel 5.2.1 aufgeführt.

Sowohl die virtuellen Maschinen auf der CPU als auch die auf der GPU haben im Vergleich zur Hardwareimplementierung auf dem FPGA eine deutlich niedrigere Performance (siehe Tabelle 5.3). Der Beschleunigungsfaktor durch eine FPGA-Implementierung der Zellularautomaten liegt im Vergleich zur jeweils schnellsten Software-Auswertung zwischen 957 (Vergleich zu `sumEuler` - CPU) und 6.278 (Vergleich zu `sumSqrtLU` - GPU) für die untersuchten Algorithmen.

Die Anzahl an benötigten Generationen bis zur vollständigen Auswertung der Algorithmen ist für den FPGA, verglichen mit der Softwarelösung, um den Faktor 3,8 bis 28,5 geringer. Diese Differenz ergibt sich einerseits aus zusätzlichen Optimierungen durch die den VHDL-Code auf den FPGA abbildende Software. Andererseits führen die in Kapitel 3.7.2 beschriebenen Verfahren, wie zum Beispiel selbstsynchronisierende Kombination und Weiterleitung von Signalen, zu einer Reduktion der Anzahl an Zellgenerationen.

Die möglichen Taktraten zur Umsetzung der Zellularautomaten auf die spezifische FPGA-Architektur liegen bei 222,2 MHz für die Algorithmen `gcd` und `sumSqrt`, bei 192,2 MHz für Algorithmus `sumEuler` sowie bei 2,1 MHz für Algo-

rithmus `sumSqrtLU`. Der niedrige erreichbare Takt für Algorithmus `sumSqrtLU` erklärt die Differenz zwischen den Verhältnissen von Zellgenerationen zu Laufzeiten im Vergleich zu den ersten drei Algorithmen.

Während die Laufzeiten auf der CPU und GPU von vielen Faktoren, zum Beispiel der Anzahl an zur Verfügung stehender Prozessoren sowie dem Scheduler des Systems abhängt, wird die Laufzeit auf dem FPGA direkt von der Anzahl an Zellgenerationen und der Taktrate beeinflusst. Dieser Umstand resultiert aus der vollständigen parallelen Bearbeitung aller im Zellularautomaten enthaltenen Zellen auf dem FPGA. Gleichzeitig bedeutet das, dass die langsamste Zelle sowie die notwendigen Verbindungen zwischen den Zellen den Systemtakt vorgeben. Je mehr Zellen ein Zellularautomat beinhaltet, desto mehr Fläche und Kommunikationswege werden auf dem FPGA benötigt, was zu einer niedrigeren Taktfrequenz führt [31].

Das Experiment hat gezeigt, dass bei der Auswertung der Zellularautomaten die Performance des ausführenden Systems stark von den zu kompilierenden Algorithmen und der Anzahl der daraus erzeugten Zellen abhängt. Während die CPU bei kleinen Zellularautomaten (weniger als 1000 Zellen) der GPU in ihrer Performance überlegen ist, so ist bei hinreichend großen Zellularautomaten die GPU im Vorteil, da mehr Rechenkerne zur Verfügung stehen, auf die die Zellen verteilt werden können. Weiterhin wurde durch das Experiment bestätigt, dass eine Hardwareimplementierung der Zellularautomaten deutlich performanter in Bezug auf die Laufzeit der Algorithmen verglichen mit der Softwarelösung ist.

5.2 Einfluss von Laufzeitparametern auf die Performance der virtuellen Maschinen

In diesem Kapitel werden die Unterschiede zwischen Debug- und Release-Modus der virtuellen Maschinen zur Auswertung der Zellularautomaten aufgezeigt. Gegenstand des Kapitel sind des Weiteren Untersuchungen zum Einfluss der Thread-Blockgröße sowie die Auswirkung der Zellanordnung auf die Performance der Zellularautomaten auf der GPU.

5.2.1 Experiment 4 - Vergleich zwischen Debug- und Release-Modus

Der Debug- und Release-Modus der virtuellen Maschinen zur Bearbeitung der Zellularautomaten unterscheiden sich für den Benutzer des Frameworks allein durch die im Debug-Modus sichtbare Darstellung der Zellzustände jeder Zellgeneration nach deren Auswertung. Weitere Unterschiede zu dem für den Debug-Modus angewendeten Verfahren ergeben sich, wie in Kapitel 3.6.4 beschrieben, durch die Art der Ausführung und Verwaltung der die Zellen bearbeitenden Threads im Release-Modus.

In diesem Experiment soll lediglich die Differenz der verschiedenen Ausführungsmodi, die durch die unterschiedliche Behandlung und Verwaltung der Zellen entsteht, untersucht werden. Dazu wurden die visuellen Updates und Synchronisationen zur Oberfläche des Programms im Debug-Modus für die vorgenommenen Messungen deaktiviert.

Die für das Experiment untersuchten Algorithmen und daraus generierten Zellularautomaten sind identisch zu denen aus Kapitel 5.1.3. Mit Tabelle 5.4 folgt die Gegenüberstellung der benötigten Zellgenerationen zur Auswertung der Algorithmen und mit Tabelle 5.5 die Gegenüberstellung der ermittelten Laufzeiten für die Beispiele im Debug-Modus und Release-Modus der virtuellen Maschinen. Die Werte für den Release-Modus sind aus Kapitel 5.1.3 übernommen. Die Messwerte für den Debug-Modus wurden nach dem gleichem Verfahren (Anzahl an Durchläufen zur Bildung der Mittelwerte) wie die des Release-Modus ermittelt.

Betrachtet man die Anzahl an Zellgenerationen bis zur Auswertung der Algorithmen, so ist erkennbar, dass sowohl die virtuelle Maschine auf der CPU als auch die auf der GPU im Debug-Modus weniger (bzw. gleich viele) Updates benötigt, um die vollständige Bearbeitung der Zellularautomaten abzuschließen.

Im Fall der GPU entsteht dieser Effekt durch die asynchronen Kernel-Aufrufe mit wenigen Synchronisationspunkten (nach jedem 100. Update). Die asynchronen Kernel-Aufrufe dienen dazu, die Wartezeit auf der GPU zwischen zwei Updates möglichst gering und somit die Auslastung der GPU möglichst hoch zu halten (siehe Kapitel 3.6.4). Daher ist es möglich, dass der Zellularautomat auf der GPU bereits vollständig ausgewertet ist, die bereits gestarteten Kernel-Aufrufe jedoch noch nicht abgeschlossen sind. Es

Name	gcd	sumEuler	sumSqrt	sumSqrtLU
$CPU_{Release}$	212	1.028.732	400.482	2.370
CPU_{Debug}	202	1.013.075	398.496	2.370
$GPU_{Release}$	252	1.117.822	406.038	2.500
GPU_{Debug}	221	1.117.804	406.014	2.423

Tabelle 5.4: Anzahl an benötigten Generationen zur Auswertung der Zellularautomaten in Experiment 4

werden somit Updates des Zellularautomaten durchgeführt, bei denen sich die Zustände der Zellen nicht weiter verändern, da ihre Ergebnisse bereits vorliegen.

Für die CPU-basierte Auswertung entsteht die höhere Anzahl an benötigten Zellgenerationen im Release-Modus durch den gleichzeitigen Übergang aller Threads in die nächste Zellgeneration. Das heißt, die Zellauswertung wird nahezu gleichzeitig gestartet, was die asynchrone Kommunikation (siehe Kapitel 3.6.3) der gleichzeitig gestarteten Zellen verhindert. Wird die virtuelle Maschine hingegen im Debug-Modus ausgeführt, werden nach jedem Update neue Threads zur Bearbeitung erstellt. Dadurch ist es wahrscheinlicher, dass ein Thread bereits mit der Auswertung der ihm zugewiesenen Zellen begonnen hat, bevor ein anderer Thread die Bearbeitung seiner Zellen startet. Somit können die Eingabepuffer der Zellen des einen Threads bereits durch die des anderen Threads beschrieben sein, was zu einer Reduktion der Gesamtzahl an benötigten Zellgenerationen führt. Je größer der generierte Zellularautomat ist, desto kleiner ist die Differenz der zur Auswertung nötigen Zellgeneration zwischen Debug- und Release-Modus. Sind viele Zellen durch wenige Rechenkerne zu verarbeiten, ist in beiden Ausführungsmodi asynchrone Zellkommunikation möglich, was in Tabelle 5.4 an den Messwerten für Algorithmus `sumSqrtLU` zu erkennen ist.

Die Laufzeiten der Algorithmen sind im Release-Modus trotz der größeren Anzahl an Zellgeneration zur Auswertung der Zellularautomaten kürzer, als im Debug-Modus (siehe Tabelle 5.5).

Für die ersten drei Algorithmen (`gcd`, `sumEuler` und `sumSqrt`) sind die Debug-Laufzeiten auf der CPU zwischen 8% und 18% länger, verglichen mit den Laufzeiten im Release-Modus. Die längere Laufzeit entsteht durch das

Name	gcd	sumEuler	sumSqrt	sumSqrtLU
$CPU_{Release}$	0,25 ms	1.358,71 ms	453,04 ms	770,42 ms
CPU_{Debug}	0.27 ms	1.534,45 ms	534,71 ms	776,43 ms
$GPU_{Release}$	2,46 ms	12.046,61 ms	4.072,61 ms	251,11 ms
GPU_{Debug}	8,17 ms	41.536,12 ms	14.920,43 ms	321,78 ms

Tabelle 5.5: Gesamtlaufzeiten der Zellularautomaten in Experiment 4

Neuanlegen und Zerstören der die Zellen verwaltenden Threads für jedes Update des jeweiligen Zellularautomaten. Die Differenz der Laufzeiten zwischen Debug- und Release-Modus für Algorithmus `sumSqrtLU` liegt bei 0,8%. Die vergleichsweise geringe Differenz entsteht durch das Verhältnis von Bearbeitungszeit pro Generation und Anzahl an Zellgenerationen. Je mehr Zellen pro Zellgeneration bearbeitet werden müssen, desto weniger Einfluss auf die (mit der Anzahl der Zellen steigenden) Laufzeit hat das Neuanlegen und Zerstören der Threads, da die dafür benötigte Zeit unabhängig von der Anzahl an Zellen im Zellularautomaten ist.

Bei der Auswertung der Zellularautomaten durch die virtuelle Maschine auf der GPU liegt der Beschleunigungsfaktor durch den Release-Modus verglichen mit dem Debug-Modus, für die untersuchten Algorithmen, zwischen 1,3 und 3,7. Je mehr Zellgenerationen zur vollständigen Abarbeitung des jeweiligen Algorithmus notwendig sind, desto mehr Einfluss auf die Gesamtlaufzeit haben die expliziten Synchronisationspunkte nach jedem Kernel-Aufruf im Debug-Modus. Der, verglichen mit den ersten drei Algorithmen, niedrige Beschleunigungsfaktor von circa 1,3 für Algorithmus `sumSqrtLU` entsteht durch die längere Bearbeitungszeit für ein Update des Zellularautomaten im Bezug zur Dauer eines Kernel-Aufrufes.

Die zum Debug-Modus differenzierte Verwaltung und Ausführung der für die Auswertung der Zellularautomaten zuständigen Threads im Release-Modus ist für die virtuelle Maschine auf der CPU vor allem für kleine (weniger als 1000 Zellen) Zellularautomaten von Vorteil. Mit steigender Zahl an Zellen, sinkt die Differenz zwischen Debug-und Release-Modus aufgrund des Verhältnisses zwischen wachsender Bearbeitungszeit einer Generation und der konstanten Erstellungszeit für neue Threads durch das Betriebssystem.

Ähnliche Auswirkungen einer steigenden Anzahl an Zellen auf die Laufzeit sind im durchgeführten Experiment auch für die GPU erkennbar. Die im Release-Modus verwendeten asynchronen Kernel-Aufrufe wirken sich bei den untersuchten Zellularautomaten jedoch in allen Fällen deutlich (mindestens 30% Steigerung) auf die Performance verglichen mit dem Debug-Modus aus.

5.2.2 Experiment 5 - Auswirkung unterschiedlicher Blockgrößen auf die Laufzeit der GPU

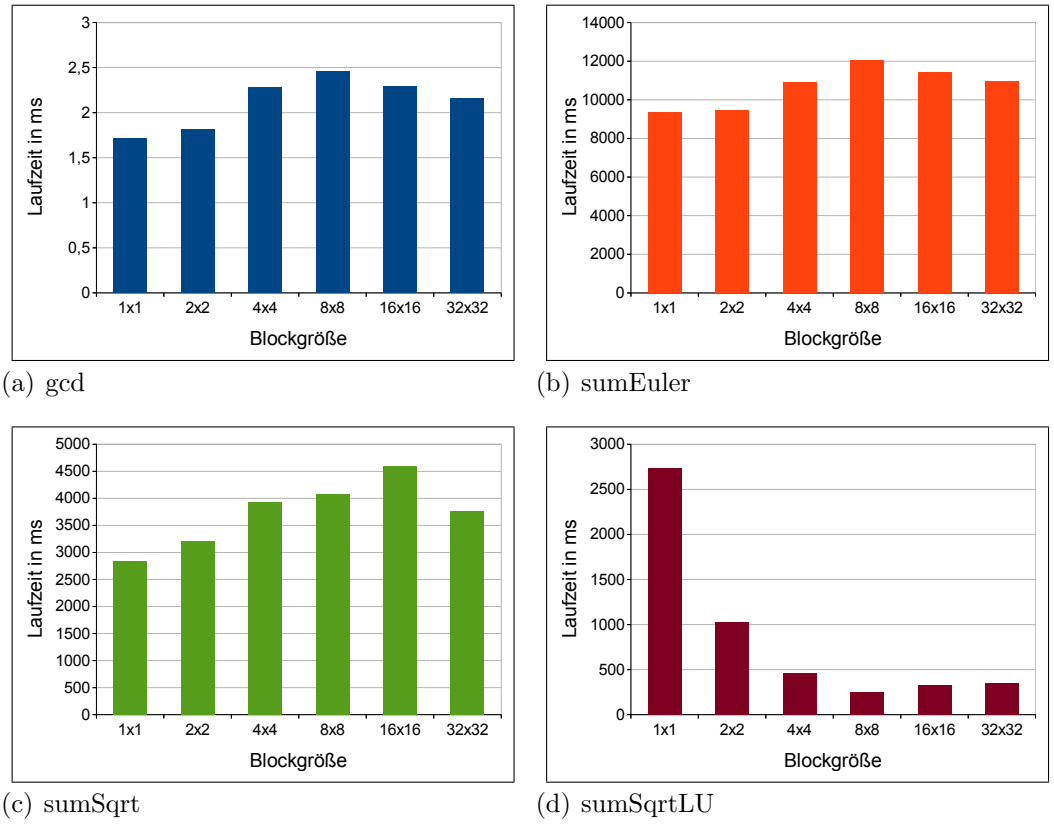
Für die virtuelle Maschine auf der CPU werden die Zellen des auszuwertenden Zellularautomaten auf alle zur Verfügung stehenden Threads aufgeteilt. Im Gegensatz dazu erfolgt für die Bearbeitung des Zellularautomaten durch die GPU die Erstellung so vieler Threads, wie Zellen im Zellularautomaten vorhanden sind. Zur anschließenden Verarbeitung durch die GPU werden diese Threads wiederum zu Blöcken zusammengefasst (siehe Kapitel 3.6.4).

In diesem Experiment wird die Auswirkung der gewählten Blockgröße auf die Laufzeit der Algorithmen untersucht. Die dazu eingesetzten Zellularautomaten sind aus den in Kapitel 5.1.3 vorgestellten Algorithmen generiert (`gcd`, `sumEuler`, `sumSqrt` und `sumSqrtLU`). Die Blockgröße und Dimension der Thread-Blöcke wird bei jedem Aufruf des Update-Kernels an die GPU übermittelt und die Threads entsprechend eingeteilt.

Abbildung 5.6(a-d) stellt die Laufzeiten der Algorithmen für verschiedene Blockgrößen gegenüber. Für das Experiment wurden quadratische Aufteilungen mit $2^n \cdot 2^n \{n \in \mathbb{N}_0 \wedge n \leq 5\}$ Threads verwendet. Die maximale Größe von 1024 Threads innerhalb eines Blockes ist durch die verwendete Hardware beschränkt.

In den Diagrammen ist erkennbar, dass die momentan für das Framework verwendete Blockgröße von 64 (8x8) Threads für die ersten drei Algorithmen, mit einer niedrigen Anzahl an Zellen, keine optimalen Laufzeiten ergibt. Für die Algorithmen `gcd` (Abbildung 5.6(a)) und `sumEuler` (Abbildung 5.6(b)) hat eine Aufteilung von 8x8 Threads pro Block sogar die längste Laufzeit verglichen mit den anderen untersuchten Blockgrößen.

Die Auswertung der Algorithmen ist bei den ersten drei Algorithmen optimal, wenn lediglich ein Thread pro Block verwendet wird. Die dadurch auf der GPU gebildeten Warps (siehe Kapitel 3.6.4) werden so ausgeführt, als ob



(a) gcd (b) sumEuler

(c) sumSqrt (d) sumSqrtLU

Abbildung 5.6: Laufzeiten zur Auswertung von Zellularautomaten durch die GPU für unterschiedliche Blockgrößen

32 Threads zur Verarbeitung vorhanden wären, obwohl die Berechnung von lediglich einem Thread relevant ist. Das heißt für die eingesetzte GPU mit 512 Rechenkernen, dass (bei einem Thread pro Block) 16 Zellen parallel ausgewertet werden. Bei Zellularautomaten mit wenigen und somit in ihrer Berechnung stark voneinander abweichenden Zellen (Verhältnis von Anzahl an Zelltypen und Anzahl an Zellen insgesamt) führt die Verwendung von kleinen Blöcken (weniger als 8x8 Threads) zu einer Steigerung der Performance. Das wird möglich, da innerhalb eines Warps somit wenige (Blockgröße > 1) bzw. gar keine (Blockgröße $= 1$) Synchronisationspunkte notwendig sind, weil alle Threads eines Warps die gleichen Instruktionen ausführen.

Eine höhere Anzahl an Threads pro Block führt dazu, dass die Berechnungen innerhalb eines Warps mit den Daten der verschiedenen Zellen des Zellularautomaten durchgeführt werden. Für die ersten drei Algorithmen mit einer im

Verhältnis zur Gesamtgröße des Zellularautomaten großen Anzahl verschiedener Zelltypen bedeutet das, dass innerhalb eines Warps kaum parallele Verarbeitung möglich ist und die Abarbeitung der Zellen serialisiert vorgenommen wird. Dieser zusätzliche Verwaltungsoverhead führt zu einer längeren Laufzeit verglichen mit der Aufteilung auf kleine Thread-Blöcke.

Abbildung 5.6(d) zeigt das umgekehrte Laufzeitverhalten für Algorithmus `sumSqrtLU`. In diesem Fall ist eine Blockgröße von 8x8 Threads optimal für die Performance der virtuellen Maschine, da innerhalb eines Warps viele gleich zu verarbeitende Zellen enthalten sind, die mit wenigen Synchronisationspunkten auskommen. Eine noch höhere Anzahl an Threads pro Block (im Beispiel ab 256) ist jedoch auch hier nicht vorteilhaft. Dieser Effekt entsteht durch die begrenzte Anzahl an pro Block zur Verfügung stehenden Registern. Sind zu viele Threads innerhalb eines Blocks enthalten, können Daten nicht mehr in Registern gespeichert werden, was sich negativ auf die Dauer von Berechnungen auswirkt [25].

Eine hohe Anzahl an zur Verfügung stehenden Rechenkernen auf der GPU führt nicht automatisch zu einer schnelleren Verarbeitung. Die Einteilung der Threads in Blöcke hat einen starken Einfluss auf die Laufzeit der Algorithmen. Problematisch hierbei ist die Individualität der kompilierten Zellularautomaten, die eine Vorhersage und optimale Bestimmung der Blockgröße für alle Anwendungsfälle verhindert.

Für Weiterentwicklungen des Systems ist ein dynamisch adaptierender Algorithmus während der Auswertung der Zellularautomaten denkbar, der das Laufzeitverhalten im Bezug zur Blockgröße analysiert und diese entsprechend verändert.

5.2.3 Experiment 6 - Einfluss der Zellenanordnung auf die Laufzeiten der GPU

Neben der in Experiment 5 (siehe Kapitel 5.2.2) untersuchten Blockgröße hat die Zellanordnung innerhalb des zweidimensionalen Gitters ebenfalls einen Einfluss auf die Anzahl der Zellgenerationen und die Laufzeiten der Algorithmen auf der GPU. Die Auswirkungen einer Änderung der Anordnung wurden in diesem Experiment untersucht und sind im Folgenden beschrieben.

Name	gcd	sumEuler	sumSqrt	sumSqrtLU
Gen_{Exp3}	252	1.117.822	406.038	2.500
$\text{Gen}_{\text{Sortiert}}$	287	1.156.400	453.036	2.499
$\text{Zeit}_{\text{Exp3}}$	2,46 ms	12.046,61 ms	4.072,61 ms	251,11 ms
$\text{Zeit}_{\text{Sortiert}}$	2,82 ms	12.014,72 ms	4.183,02 ms	223,26 ms

Tabelle 5.6: Gesamtlaufzeiten und benötigte Generationen in Experiment 6

Als Referenz dienen die in Experiment 3 in Kapitel 5.1.3 vorgestellten Zellularautomaten und deren Laufzeiten sowie die Zellgenerationen zur Auswertung der Algorithmen. Die für dieses Experiment ermittelten Vergleichswerte wurden durch die Auswertung der selben Zellularautomaten gebildet. Die Zellen wurden jedoch nicht durch Tiefentraversierung aus der Baumstruktur in das zweidimensionalen Gitter übertragen (siehe Kapitel 3.6.1), sondern nach Zelltypen sortiert abgebildet.

Tabelle 5.6 stellt die gemessenen Laufzeiten und Zellgenerationen zur Auswertung der Algorithmen aus Experiment 3 (Exp3) und die anhand der sortierten Zellularautomaten (Sortiert) ermittelten gegenüber.

Während sich bei den ersten drei (vergleichbar kleinen) Zellularautomaten die Sortierung der Zellen kaum bemerkbar (`sumEuler`) bzw. negativ (`gcd`, `sumSqrt`) auf die Anzahl an Zellgenerationen und die Laufzeiten der Algorithmen auswirkt, so hat die Sortierung einen positiven Einfluss auf den aus Algorithmus `sumSqrtLU` generierten Zellularautomaten. Die Anzahl an Zellgenerationen ist nahezu identisch, dagegen hat sich die Laufzeit zur Auswertung des Zellularautomaten um circa 11% reduziert. Dieser Effekt entsteht durch die in Kapitel 3.6.4 beschriebenen Eigenschaften der Grafikprozessoren, die darauf optimiert sind identische Befehle innerhalb eines Warps auszuführen.

Die Sortierung der Zellen führt bei Algorithmus `sumSqrtLU` dazu, dass die 8x8 großen Threadblöcke auf der GPU (größtenteils) identische Zelltypen enthalten. Zusätzlich arbeiten bei diesem Zellularautomaten viele Zellen parallel (siehe Kapitel 5.1.3), so dass die Bedingungen für eine optimale Warp-Ausführung erfüllt sind. Hingegen führt die (im Verhältnis zur Gesamtgröße) hohe Anzahl an verschiedenen Zelltypen und in unterschiedlichen Zellgenerationen durchgeführte Auswertung der ersten drei Algorithmen zu einer serialisierten und somit langsamen Warp-Abarbeitung.

Eine Änderung der Anordnung der Zellen innerhalb des zweidimensionalen Gitters kann sich positiv auf die Laufzeiten und Anzahl an benötigten Generationen zur Auswertung von Algorithmen auswirken. Die Voraussetzung für einen positiven Einfluss ist eine Sortierung die dafür sorgt, dass identische Zelltypen, deren Auswertung zudem in der selben Zellgenerationen stattfindet, innerhalb eines Warps bearbeitet werden.

Sind die Zellularautomaten verhältnismäßig klein (verglichen mit der Anzahl der enthaltenen unterschiedlichen Zelltypen), kann sich eine Sortierung aufgrund der zu verschiedenen Zeitpunkten durchgeführten Auswertung der Zellen auch negativ auf die Laufzeit und die Anzahl an benötigten Zellgenerationen auswirken.

5.3 Experimente zur hardwarebasierten Auswertung

Dieses Kapitel befasst sich explizit mit der Bearbeitung der Zellularautomaten durch einen FPGA (siehe Einleitung Kapitel 5 für weitere Informationen zum eingesetzten FPGA). Wie anhand der vorangegangenen Untersuchungen erkennbar wird, ist die hardwarebasierte Auswertung der Zellularautomaten den softwarebasierten Varianten im Bezug auf ihre Performance deutlich überlegen.

Im Folgenden werden die Unterschiede zwischen taktsynchronisierter und selbstsynchronisierender Verarbeitung der Zellen aufgezeigt sowie die Ergebnisse eines Performancevergleichs zwischen einem durch einen Standardcompiler übersetzten C-Programms zu einem, den selben Algorithmus repräsentierenden, Zellularautomaten dargestellt.

5.3.1 Experiment 7 - Vergleich zwischen taktsynchronisierter und selbstsynchronisierender Verarbeitung

Wie in Kapitel 3.7.2 beschrieben, wird bei der Implementierung der Zellen auf dem FPGA teilweise selbstsynchronisierende Verarbeitung eingesetzt, um eine Optimierung bezüglich der Geschwindigkeit und des notwendigen Platzbedarfs für die Abbildung der Zellen in Hardware zu erreichen. In diesem Experiment wurden die Auswirkungen einer selbstsynchronisierenden Verarbeitung im Vergleich zu einer vollständig taktsynchronisierten Verarbeitung der Zellen untersucht.

Name	gcd	sumEuler	sumSqrt	sumSqrtLU
Reg_S	673	817	345	9.392
Reg_T	1.345	1.912	744	-
LUT_S	600	1.330	411	23.915
LUT_T	606	919	491	-
Takt_S	222,22 MHz	192,31 MHz	222,22 MHz	2,17 MHz
Takt_T	250,00 MHz	263,16 MHz	270,27 MHz	-
Gen_S	54	272.942	85.004	83
Gen_T	159	856.553	279.007	1.269
Zeit_S	$2,43 \cdot 10^{-4}$ ms	1,42 ms	0,38 ms	0,04 ms
Zeit_T	$6,36 \cdot 10^{-4}$ ms	3,25 ms	1,03 ms	-

Tabelle 5.7: Anzahl an notwendigen Registern (Reg), Lookup-Tables (LUT), erreichbare Taktfrequenz (Takt) und Zellgeneration (Gen) sowie Gesamtlaufzeiten (Zeit) der Zellularautomaten in Experiment 7

Als Vergleichsgrundlage dienen die Algorithmen aus Experiment 3 (siehe Kapitel 5.1.3). Tabelle 5.7 stellt die ermittelten Werte der Zellularautomaten mit taktsynchronisierter (T) und selbstsynchronisierender (S) Verarbeitung gegenüber. Die Werte beziehen sich auf die finale Implementierung auf dem FPGA nach *Place & Route*. Für Algorithmus `sumSqrtLU` konnten keine Realwerte zur tatsächlichen Größe und Geschwindigkeit bei taktsynchronisierter Verarbeitung aufgezeichnet werden, da die Berechnung zu umfangreich für das Synthesetool ist. Somit enthält die Tabelle für Algorithmus `sumSqrtLU` bei taktsynchronisierter Verarbeitung lediglich die durch Simulation bestimmte Anzahl an notwendigen Zellgenerationen zur Auswertung des Algorithmus.

Anhand der Tabelle ist ersichtlich, dass die Abbildung der selbstsynchronisierenden Zellen weniger (Faktor 2 bis 2,3) Register auf dem FPGA benötigt als die Zellen, die ausschließlich über den Takt synchronisiert werden. Im Falle der selbstsynchronisierenden Zellen werden durch das Synthesetool sowie bei anschließenden Optimierungen während des *Mappings* bzw. *Place & Route* unnötige Zwischenpuffer entfernt, was bei einer taktsynchronisierten Verarbeitung nicht möglich ist.

Die Anzahl an notwendigen Lookup-Tables ist für Algorithmus `gcd` bei beiden Verarbeitungsmodi nahezu identisch. Sie ist für Algorithmus `sumEuler` bei selbstsynchronisierender Verarbeitung um den Faktor 1,4 höher und für

Algorithmus `sumSqrt` bei selbstsynchronisierender Verarbeitung wiederum um den Faktor 1,2 geringer. Die Varianz bezüglich der Größe entsteht durch die Unterschiedlichkeit der Algorithmen. Werden viele Zwischenergebnisse bei der selbstsynchronisierenden Verarbeitung entfernt, kann das zu komplexeren Operationen führen, so dass aufwändigere (mehrere hintereinander geschaltete) Lookup-Tables notwendig sind, um diese zu realisieren. Darüber hinaus wird ermöglicht, dass durch die selbstsynchronisierende Verarbeitung keine zusätzlichen komplexen Operationen notwendig sind bzw. diese durch weitere Optimierungen aufgelöst werden, so dass weniger Lookup-Tables als bei taktsynchronisierter Verarbeitung benötigt werden.

Die durch die selbstsynchronisierende Verarbeitung möglichen Optimierungen und das Zusammenfassen mehrerer Operationen innerhalb eines Taktzyklus führen zu einer verringerten Anzahl an notwendigen Zellgenerationen zur Auswertung der Algorithmen (siehe Tabelle 5.7). Für die ersten drei Algorithmen ist die Anzahl an notwendigen Zellgenerationen um den Faktor ≈ 3 geringer als bei taktsynchronisierter Verarbeitung. Der Wert für Algorithmus `sumSqrtLU` ist darüber hinaus um den Faktor ≈ 15 geringer als der durch Simulation errechnete Wert für die taktsynchronisierte Verarbeitung der Zellen. In diesem speziellen Fall entsteht die vergleichsweise hohe Differenz durch den `+=`-Operator, der am Ende des Algorithmus zur Bildung der Summe der Einzelberechnungen verwendet wird. Bei taktsynchronisierter Verarbeitung müssen die insgesamt 1.000 Berechnungen nacheinander durchgeführt werden, während bei selbstsynchronisierender Verarbeitung die Summe (bei gleichzeitigem Vorliegen aller Zwischenergebnisse) innerhalb eines Taktzyklus gebildet werden kann.

Der Nachteil der selbstsynchronisierenden Verarbeitung ist die damit verbundene niedrigere (stabil) erreichbare Taktfrequenz. Dies ist insbesondere an dem aus Algorithmus `sumSqrtLU` generierten Zellularautomaten erkennbar. Die Taktfrequenz liegt in diesem Fall bei lediglich 2,17 MHz und begründet sich durch die parallele Durchführung des `+=`-Operators sowie der im Vergleich zu den anderen Algorithmen hohen Anzahl an Registern und Lookup-Tables, was zu längeren bzw. komplexeren und somit langsameren Verbindungswegen auf dem FPGA führt [31]. Für die Algorithmen `gcd`, `sumEuler` und `sumSqrt` sind die für die selbstsynchronisierende Verarbeitung erreichbaren Taktfrequenzen,

verglichen mit der taktsynchronisierten Verarbeitung, um den Faktor 1,13 bis 1,37 niedriger.

Die geringfügig niedrigere Taktfrequenz in Kombination mit der deutlich geringeren Anzahl an notwendigen Zellgenerationen zur Auswertung der Algorithmen führt zu einem Performancevorteil im Bezug auf die Gesamtlaufzeiten der Zellularautomaten. Diese sind für die Zellen mit selbstsynchronisierender Verarbeitung, verglichen mit den Gesamtlaufzeiten der Zellen, die durch taktsynchronisierte Verarbeitung ausgewertet wurden, um den Faktor 2,29 bis 2,71 geringer.

Fazit der experimentell durchgeführten Untersuchung ist, dass eine (teilweise) selbstsynchronisierende Verarbeitung zur Auswertung der Zellularautomaten sowohl im Bezug auf die Größe der Zellen auf dem FPGA als auch im Bezug auf die Laufzeiten der Algorithmen Vorteile bietet. Allerdings wird auch deutlich, dass weiterführende Forschungen zur besseren Aufteilung der Bereiche, auf die die selbstsynchronisierende Verarbeitung angewendet wird, notwendig sind. Eine zu starke Bündelung der Berechnungen innerhalb eines Taktzyklus führt zu einer erheblich verringerten (stabil) erreichbaren Taktfrequenz und sollte daher vermieden werden.

5.3.2 Experiment 8 - Performancevergleich zwischen Zellularautomaten und CPU-Software

Das im Folgenden beschriebene Experiment vergleicht die Performance zwischen Algorithmen, kompiliert für die CPU durch einen Standardcompiler (Microsoft Visual Studio 2010 [75]), und den selben Algorithmen, übersetzt als Zellularautomaten auf den für die Experimente zur Verfügung stehenden FPGA (Details zur CPU und dem FPGA sind in der Einleitung zu Kapitel 5 aufgeführt).

Die für die Untersuchung eingesetzten Algorithmen sind in Abbildung 5.7 dargestellt. Beim ersten Algorithmus (siehe 5.7(a)) handelt es sich um eine *Bubble-Sort*-Implementierung [69], die dazu eingesetzt wird, einen Testdatensatz (Array `A[]`) mit 100 Werten zu sortieren. Die Werte innerhalb des Arrays wiederholen sich nach 10 Elementen. Die Korrektheit der `bubbleSort`-Funktion wird durch Aufsummierung des Fehlers `error` (falsche Position eines Elements) am Ende der Hauptfunktion überprüft.

```
int A[] = {
 23, 34, 5, 11, 1, 8, 77, 4, 15, 9,
 23, 34, 5, ... };

int numP = 100;

void bubbleSort()
{
  int j, i, temp;

  for ( j = numP; j > 1; j-- )
  {
    for ( i = 0; i < j-1; i++ )
    {
      if ( A[ i ] > A[ i+1 ] )
      {
        temp = A[ i ];
        A[ i ] = A[ i+1 ];
        A[ i+1 ] = temp;
      }
    }
  }
}

int bubble()
{
  int i, error = 0;

  bubbleSort();

  for ( i = 1; i < numPoints; i++ )
    if ( A[ i ] < A[ i-1 ] )
      error++;

  return error;
}
```

(a) bubble.c

```
int A[] = { 1680,  2016,  2688,
            3528,  3780, 14175,
           16380, 23100, 24696,
           31500 };

int numP = 10;

int gcd_loop()
{
  int sum = 0;
  int i = 0, j = 0;

  for ( i = 0; i < numP; i++ )
    for ( j = 0; j < numP; j++ )
      sum += gcd( A[ i ], A[ j ] );

  return sum;
}
```

(b) gcd_loop.c

```
int decrypt( int c, int key )
{
  return ( c ^ key );
}

int find_key( int c, int p )
{
  int key = 0;

  while( decrypt( c, key ) != p )
    key++;

  return key;
}
```

(c) find_key.c

Abbildung 5.7: C-Code für Experiment 8

Der zweite Algorithmus (siehe Abbildung 5.7(b)) verwendet den in Experiment 3 beschriebenen `gcd`-Algorithmus (siehe Kapitel 5.1.3 bzw. [98]) zur Bildung der Summe des paarweise berechneten *größten gemeinsamen Teilers* von 10 verschiedenen Zahlen. Der Algorithmus wurde einmal mit und einmal ohne Loop-Unrolling der äußeren `for`-Schleife bei der Erzeugung des Zellularautomaten übersetzt.

Der dritte Algorithmus (siehe Abbildung 5.7(c)) ermittelt durch eine *Brute-*

Name	bubble	gcd_loop	gcd_loop (LU)
Zellen	1.641	192	1.446
$\text{Takt}_{\text{FPGA}}$	131,58 MHz	232,56 MHz	120,48 MHz
Takt_{CPU}	3.400,00 MHz	3.400,00 MHz	3.400,00 MHz
$\text{Zeit}_{\text{Zellularautomat}}$	0,262 ms	0,027 ms	0,006 ms
$\text{Zeit}_{\text{Software}}$	0,023 ms	0,011 ms	0,011 ms
$\text{Faktor}_{\text{real}}$	0,09	0,41	1,83
$\text{Faktor}_{\text{taktbereinigt}}$	2,27	5,96	51,74

Tabelle 5.8: Anzahl an Zellen, Taktfrequenz und Laufzeiten sowie Beschleunigungsfaktor bei Verarbeitung durch Zellularautomaten bzw. durch CPU-Software in Experiment 8

Force-Suche [84] den zur Verschlüsselung einer Zahl `p` (bis zu 31-Bit) eingesetzten Schlüssel `key` bei Kenntnis der verschlüsselten Zahl `c`. Die Verschlüsselung wird im Beispiel durch eine **xor**-Verknüpfung realisiert.

Die experimentellen Ergebnisse der Ausführung der Algorithmen in Software bzw. als Zellularautomat auf dem FPGA sind für die ersten beiden Algorithmen in Tabelle 5.8 dokumentiert. Zur besseren Vergleichbarkeit wurden die *Turbo Boost*- als auch die Energiesparfunktion des Prozessors ausgeschaltet, so dass die eingesetzte CPU dauerhaft mit 3,4 GHz getaktet ist. Damit sowohl der Zellularautomat als auch die Software grundsätzlich die gleichen Befehle ausführen, wurden die Algorithmen in beiden Fällen ohne Optimierungen kompiliert.

Betrachtet man die Laufzeiten der einzelnen Algorithmen, so benötigt die Software nur 41% der Zeit zur Auswertung von Algorithmus `gcd_loop` im Vergleich zum Zellularautomaten auf dem FPGA. Für Algorithmus `bubble` ist die Berechnung der Software in 9% der Laufzeit des Zellularautomaten abgeschlossen. Bei Anwendung von Loop-Unrolling und damit Erhöhung des Parallelitätsgrades ist trotz niedrigerer erreichbarer Taktfrequenz die Auswertung des Zellularautomaten, verglichen mit der Softwareimplementierung, um den Faktor 1,83 performanter.

Verschiedene Laufzeiten für unterschiedliche Eingabewerte des `brute_force`-Algorithmus sind in Abbildung 5.8 dargestellt. Die Laufzeiten des FPGA enthalten für dieses Experiment zusätzlich zur Auswertung des Algorithmus

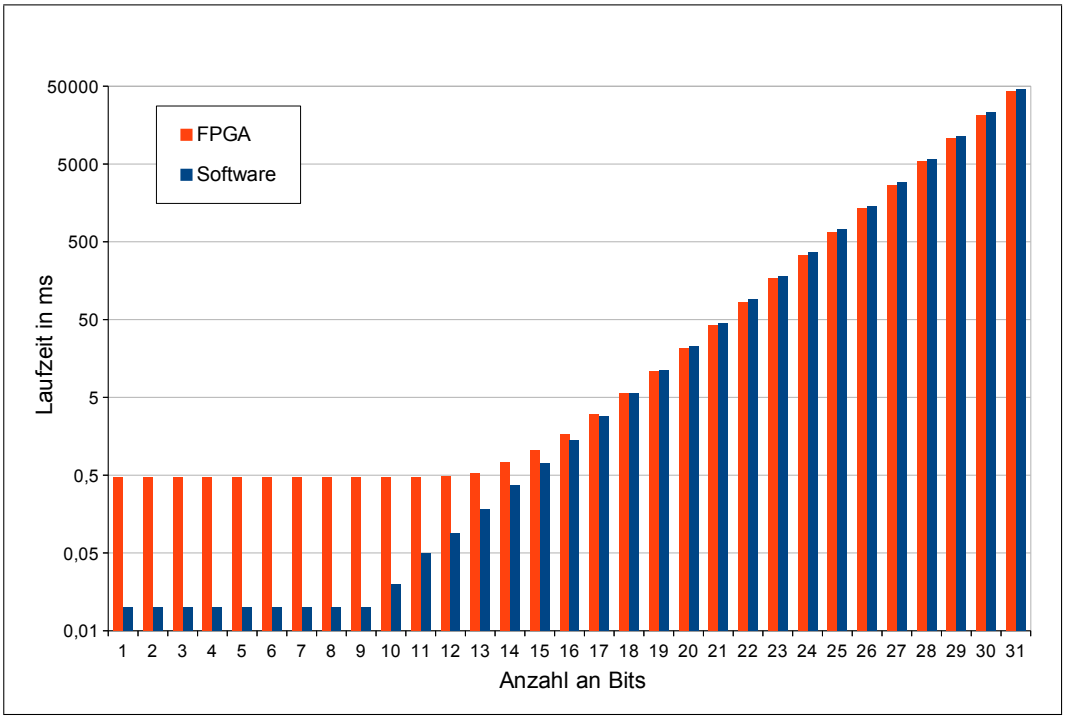


Abbildung 5.8: Laufzeiten des `brute_force`-Algorithmus in kompilierter CPU-Software und als Zellularautomat auf einem FPGA

die Übertragungszeit zur Benutzeroberfläche am PC, die die Ergebnisse vom FPGA erhält und für den Benutzer anzeigt. Die Eingabewerte stellen für die jeweilige Anzahl an Bits n den *Worst-Case*-Fall dar. Das heißt, die verschlüsselte Zahl entspricht $2^n - 1$, die entschlüsselte Zahl 0. Somit müssen alle Varianten von 0 bis $2^n - 1$ getestet werden, bevor der korrekte Schlüssel gefunden wird.

Im Diagramm ist erkennbar, dass die Laufzeit des FPGA für niedrige Bitbreiten (bis 17 Bit) länger ist als die Laufzeit der CPU-Software. Dieser Effekt entsteht durch die, im Vergleich zur Berechnungszeit, lange Übertragungszeit der Eingabewerte bzw. Ergebnisse über die Ethernet-Schnittstelle des PC bzw. FPGA (siehe Kapitel 3.7.3). Für Werte ab 18 Bit ist der FPGA wiederum performanter als die Software, da die konstante Übertragungszeit mit zunehmender Berechnungszeit weniger Einfluss auf die Gesamtlaufzeit hat.

Die gemessenen Laufzeiten zwischen FPGA und Software unterscheiden sich nur geringfügig. Bei 31 Bit benötigt die Software ca. 46 Sekunden für die Berechnung, der FPGA benötigt ca. 43 Sekunden. Das entspricht einem Beschleu-

nigungsfaktor von 1,07. Die niedrige Taktfrequenz von 200 MHz für den FPGA bei Abbildung des Algorithmus ergibt taktbereinigt jedoch einen Beschleunigungsfaktor von $\approx 18{,}25$.

Das Experiment hat gezeigt, das eine reale Beschleunigung eines Programms durch die Übersetzung in einen Zellularautomaten und Abbildung auf einen FPGA, verglichen mit einer Softwareimplementierung, nicht automatisch gegeben ist. Das Ergebnis ist von der möglichen Parallelität innerhalb des übersetzten Algorithmus abhängig. Verhindern Datenabhängigkeiten die parallele Ausführung, so ist eine Softwareimplementierung performanter als ein zur Auswertung des selben Algorithmus eingesetzter Zellularautomat.

Taktbereinigt ist der eingesetzte FPGA der Softwarelösung jedoch in allen Fällen überlegen, was eine hardwarebasierte Auswertung der Zellularautomaten für schnelle und gleichzeitig energieeffiziente Bearbeitung von Algorithmen interessant macht.

6 Zusammenfassung

Die vorliegende Arbeit stellt eine neue Herangehensweise bei der Abbildung sequentieller Programme auf parallele Architekturen vor. Während existierende Ansätze auf bestimmte Zielarchitekturen und damit verbundene spezifische Lösungen konzentriert sind, basiert das vorgestellte System auf einer auf Zellularautomaten aufbauenden Programmrepräsentation, die universell für verschiedene Architekturen eingesetzt werden kann.

Grundlegende Begriffsdefinitionen sowie verschiedene Parallelisierungstechniken sind Gegenstand von Kapitel 2. Dort werden frühe Techniken zur Parallelisierung auf Befehlsebene vorgestellt, so zum Beispiel Pipelining und Auslagerung von Berechnungen in funktionale Einheiten. Darauf folgt die Beschreibung verschiedener Supercomputer, des *Moorschen* und *Amdahlschen Gesetzes* sowie der Einordnung von Architekturen in die *Flynnsche Klassifikation*. Anschließend erläutert Kapitel 2 existierende Parallelisierungstechniken für verschiedene parallele Architekturen (CPU, GPU und FPGA), ihre möglichen Einsatzgebiete und Einschränkungen. Des Weiteren werden klassische und zur Lösung allgemeiner Aufgaben entwickelte Zellularautomaten vorgestellt.

Kapitel 3 beschreibt die Umwandlung von Algorithmen in Zellularautomaten - vom Einlesen des Quellcodes und Erzeugung der Zellen bis zur Ausführung auf verschiedenen parallelen Architekturen. Neben der Auswertung der Zellularautomaten in Software beinhaltet das Kapitel zusätzlich die Implementierungsdetails eines hardwarebasierten Ansatzes, der die generierten Zellen auf einen FPGA abbildet und die Ergebnisse der Algorithmen an die Anwendersoftware überträgt. Es werden Unterschiede zu bereits existierenden automatisch parallelisierenden Verfahren aufgezeigt sowie bereits umgesetzte und zukünftig mögliche Optimierungen erläutert.

Kapitel 4 beschreibt Erweiterungen des Basis-Systems um zusätzliche, auf Zellularautomaten abbildbare, C-Konstrukte. Neben der Realisierung von Zellen mit verschiedenen Datentypen, globalen Variablen und Funktionen der C-

Standardbibliothek am Beispiel von `printf` werden Methoden zur Abbildung von Arrays und Pointern auf Zellularautomaten dargestellt.

Die Ergebnisse verschiedener experimentell durchgeführter Untersuchungen enthält Kapitel 5. In einem ersten Experiment erfolgte die Überprüfung der automatischen parallelen Ausführung unabhängiger Schleifen. Anschließend wurden der Effekt von Loop-Unrolling mit verschiedenen Strategien sowie die Performance zwischen der Auswertung auf der CPU, der GPU und dem FPGA untersucht. Es folgen Erläuterungen zum Einfluss von Laufzeitparametern auf die Bearbeitung der Zellularautomaten sowie Vergleiche zwischen einem C-Programm, übersetzt durch einen Standardcompiler, und einem Zellularautomaten, der auf einen FPGA abgebildet wurde. Insgesamt stellte sich die Hardwarelösung auf dem FPGA als performantestes System heraus. Nachteilig ist lediglich der, verglichen mit der Softwarelösung, aufwändige Kompilierungsprozess (Codegenerierung und Hardwareabbildung durch externe Software) sowie die Größeneinschränkung der auf den FPGA abbildbaren Zellularautomaten.

7 Ausblick

Neben Erweiterungen des Systems zur Unterstützung der vollständigen C-Syntax sollte in Weiterentwicklungen die Parallelität der übersetzten Programme durch Auflösung von Datenabhängigkeiten erhöht werden. Einen möglichen Ansatzpunkt für solche Optimierungen bilden die einen Großteil der Berechnungszeit von Algorithmen einnehmenden Schleifen. Verfahren, wie erweitertes Loop-Unrolling, -Splitting und -Fusion, sind bekannte Transformationstechniken zur Steigerung der Parallelität von Schleifen und sind unter anderem in [41] bzw. in [113] beschrieben.

Eine Ergänzung des in Kapitel 3.3.3 erläuterten Dead-Code-Elimination Verfahrens oder die im *PEGASUS Projekt* (siehe Kapitel 2.1.2) eingesetzte spekulative Codeausführung und *lenient evaluation* bieten sich für Implementierungen zur Steigerung der Performance in Weiterentwicklungen des vorgestellten Systems an.

Die Hardwareimplementierung der Zellen auf einem FPGA kann um ein dynamisch nachladbares System erweitert werden. Anstatt die Zellen für jeden Algorithmus einzeln in Hardware abzubilden, könnten allgemeine Zellen auf den FPGA übersetzt werden und die Zellen der kompilierten Zellularautomaten in einen Speicher auf dem FPGA abgelegt werden. Zur Auswertung der Zellularautomaten lassen sich anschließend durch die allgemeinen Zellen die Zelldaten aus dem Speicher auslesen und entsprechend des geladenen Zelltyps und der enthaltenen Werte bearbeiten. Der aufwändige Übersetzungsprozess für den FPGA mit *Synthese*, *Mapping* und *Place & Route* wäre somit nur einmal für jeden spezifischen FPGA notwendig.

Ein solches Vorgehen macht zudem die Erzeugung von neuen Zellen zur Laufzeit möglich. Dadurch ist beispielsweise die Umsetzung von rekursiven Funktionen oder das dynamische Reservieren von Speicherbereichen (`malloc`, `realloc`, `calloc`) realisierbar.

Mit Wiedereinführung der in Kapitel 3.4.2 beschriebenen *Clear-Conditions*

ist eine effizientere Abarbeitung von Schleifen erreichbar. Werden die Zellen innerhalb einer Schleife automatisch (durch die *Clear-Conditions*) zurückgesetzt, kann das explizite Löschen der Zelldaten am Ende jeder Schleifeniteration entfallen und die nächste Iteration der Schleife kann schneller gestartet werden. Es Bedarf bei der Realisierung eines solchen Systems allerdings einer gesonderten Behandlung von *Reload*-Variablen.

Die Einfachheit der Zellen und der Kommunikationswege zwischen den Zellen erlaubt eine Aufteilung eines kompilierten Zellularautomaten auf ein verteiltes System. Dadurch lässt sich die Leistungsfähigkeit der Zellularautomaten weiter steigern. Zur Kommunikation der Zellen innerhalb eines Rechners und zwischen verschiedenen Rechner eignet sich beispielsweise der *Message Passing Interface (MPI)* Standard [38], da dabei nicht zwischen lokaler und globaler Kommunikation unterschieden werden muss.

Literaturverzeichnis

[1] J. Al-Eryani, *fpu100*, OpenCores.org, `http://opencores.org/project,fpu100` (abgerufen am 14.10.2014), 2009.

[2] R. Allen, K. Kennedy, *Automatic translation of FORTRAN programs to vector form*, ACM Trans. Program. Lang. Syst., 9(4), October 1987.

[3] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proceeding AFIPS '67 (Spring) Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, pages 483–485, 1967.

[4] M. Amini, *Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators*, PhD Thesis, MINES ParisTech, Paris, France, 2012.

[5] ANSI X3.159-1989, *Programming Language C*, 1989.

[6] B. Alpern, M. N. Wegman, F. K. Zadeck, *Detecting equality of variables in programs*, In Proceedings of the 15th SIGPLAN-SIGACT symposium on Principles of programming languages, pages 1–11, 1988.

[7] M. Aubury, I. Page, G. Randall, J. Saul, R. Watts, *Handel-C language reference guide*, Computing Laboratory, Oxford University, UK, 1996.

[8] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, *The FORTRAN automatic coding system*, In Papers presented at the February 26-28, western joint computer conference: Techniques for reliability (IRE-AIEE-ACM '57 (Western)), pages 188-198, ACM, New York, NY, USA, 1957.

[9] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, *Programming languages for distributed computing systems*, ACM Comput. Surv. 21 (3), September 1989.

[10] A. Balevic, B. Kienhuis, *KPN2GPU: an approach for discovery and exploitation of fine-grain data parallelism in process networks*, ACM SIGARCH Computer Architecture News, vol. 39(4), pages 66-71, 2011.

[11] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, *Effective automatic parallelization with Polaris*, International Journal of Parallel Programming, 1995.

[12] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, *Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model*, International Conference on Compiler Construction (ETAPS CC), Budapest, Hungary, April 2008.

[13] U. Bondhugula, A. Hartono, J. Ramanujan, P. Sadayappan, *A Practical Automatic Polyhedral Parallelizer and Locality Optimizer*, ACM SIGPLAN Programming Languages Design and Implementation (PLDI), Tucson, Arizona, June 2008.

[14] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, D. L. Slotnick, *The Illiac IV system*, Proceedings of the IEEE , vol. 60, no. 4, pages 369-388, April 1972.

[15] M. Boyer, *CUDA Kernel Overhead*, LAVA Lab CUDA Support, `http://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html` (abgerufen am 20.05.2015), 2015.

[16] H. C. Brearley, *ILLIAC II - A Short Description and Annotated Bibliography*, Electronic Computers, IEEE Transactions on 3, pages 399-403, 1965.

[17] W. Buchholz, *Planning a Computer System: Project Stretch*, McGraw-Hill, Inc., Hightstown, NJ, USA, 1962.

[18] M. Budiu, S.C. Goldstein, *Pegasus: An efficient intermediate representation*, Technical Report CMU-CS-02-107, CMU, May 2002.

[19] M. Budiu, G. Venkataramani, T. Chelcea, S.C. Goldstein, *Spatial computation*, SIGOPS Oper. Syst. Rev. 38, 5, pages 14-26, 2004.

[20] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, T. Czajkowski, *LegUp: high-level synthesis for FPGA-based processor/accelerator systems*, In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11), ACM, New York, NY, USA, pages 33-36, 2011.

[21] P. E. Ceruzzi, *A history of modern computing*, MIT Press, 2003.

[22] H.-H. Chou, W. Huang, J. A. Reggia, *The Trend cellular automata programming environment*, Transactions of the Society for Modeling and Simulation International, vol. 78(2), pages 59-75, 2002.

[23] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, S. K. Warren, *The ParaScope parallel programming environment*, Proceedings of the IEEE, vol. 81, no. 2, pages 244-263, Feb 1993.

[24] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide Version 4.2*, 2012.

[25] NVIDIA Corporation, *CUDA C Best Practices Guide*, 2014.

[26] E. W. Dijkstra, *Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60*, Technical Report 35, Mathematisch Centrum, Amsterdam, 1961.

[27] J. Drieseberg, C. Siemers, *C to Cellular Automata and execution on CPU, GPU and FPGA*, International Conference on High Performance Computing and Simulation (HPCS), pages 216-222, 2012.

[28] K. Eguro, *SIRC: An Extensible Reconfigurable Computing Communication API*, IEEE Symposium on Field-Programmable Custom Computing Machines (short paper), 2010.

[29] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, D. Burger, *Dark silicon and the end of multicore scaling*, In Computer Architecture

(ISCA), 2011 38th Annual International Symposium on, IEEE, pages 365-376, 2011.

[30] M. J. Flynn, *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers (C-21), no. 9, pages 948-960, Sept. 1972.

[31] M. J. Flynn, *Area-time-power and design effort: the basic tradeoffs in application specific systems*, Application-Specific Systems, Architecture Processors, ASAP 2005. 16th IEEE International Conference on. IEEE, 2005.

[32] M. Gardner, *The fantastic combinations of John Conway's new solitaire game "life"*, Scientific American, 223, pages 120-123, 1970.

[33] *Automatic parallelization in GCC*, GNU Compiler Collection, `https://gcc.gnu.org/wiki/AutoParInGCC` (abgerufen am 15.12.2014), 2012.

[34] D. Galloway, *The Transmogrifier C Hardware Description Language and Compiler for FPGAs*, IEEE Symposium on FPGAs for Custom Computing Machines, pages 136-144, 1995.

[35] N. Gershenfeld, D. Dalrymple, K. Chen, A. Knaian, F. Green, E. D. Demaine, S. Greenwald, P. Schmidt-Nielsen, *Reconfigurable asynchronous logic automata:(RALA)*, In ACM Sigplan Notices, Vol. 45, No. 1, pages 1-6, 2010.

[36] K. Gilles, *The semantics of a simple language for parallel programming*, In Information Processing '74: Proceedings of the IFIP Congress, Vol. 74, 1974.

[37] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys (CSUR), 23(1), pages 5-48, 1991.

[38] W. Gropp, L. Ewing, A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, Vol. 1, MIT press, 1999.

[39] M. Guy, *Square root by abacus algorithm*, `http://medialab.freaknet.org/martin/src/sqrt` (abgerufen am 17.03.2015), 1985.

[40] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam, *Maximizing Multiprocessor Performance with the SUIF Compiler*, IEEE Computer, December 1996.

[41] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, M. M. Khan, *Loop transformation recipes for code generation and auto-tuning*, In Languages and Compilers for Parallel Computing, pages 50-64, Springer Berlin Heidelberg, 2010.

[42] S. Hangal, M. S. Lam, *Tracking Down Software Bugs Using Automatic Anomaly Detection*, In Proceedings of the International Conference on Software Engineering, pages 291-301, May 2002.

[43] R. Hasti, S. Horwitz, *Using static single assignment form to improve flow-insensitive pointer analysis*, In Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, Volume 33 (5), pages 97-105, May 1998.

[44] M. Hazewinkel, *Totient function*, Encyclopedia of Mathematics, Springer, 2001.

[45] D. L. Heine, M. S. Lam, *A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector*, In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 168-181, June 2003.

[46] R. Hoffmann, K.-P. Völkmann, S. Waldschmidt, *Global Cellular Automata GCA: An Universal Extension of the CA Model*, ACRI 2000, Karlsruhe, Germany, 2000.

[47] M. Halbach, R. Hoffmann, *Implementing Cellular Automata in FPGA Logic*, IPDPS, vol. 16, page 258a, 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 15, 2004.

[48] J. Jendrsczok, P. Ediger, R. Hoffmann, *The Global Cellular Automata Experimental Language GCA-L*, Technischer Bericht, RA-1-2007, Technische Universität Darmstadt, FB Informatik, 2007.

[49] G. Holloway, M. D. Smith, *Machine SUIF Control Flow Graph Library*, Division of Engineering and Applied Sciences, Harvard University, 2002.

[50] G. Holloway, *The Machine-SUIF Static Single Assignment Library*, Division of Engineering and Applied Sciences, Harvard University, 2002.

[51] *IEEE 754-2008: Standard for Floating-Point Arithmetic*, IEEE Standards Association, 2008.

[52] *IEEE 1666-2011: IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Standards Association, 2012.

[53] *IBM System/360 Model 67 Functional Characteristics, Third Edition*, IBM publication GA27-2719-2, February 1972.

[54] *Automatic Parallelization with Intel Compilers*, Intel Corporation, 2011.

[55] *Intel Core i7-2600K Processor*, Intel Corporation, `http://ark.intel.com/de/products/52214` (abgerufen am 12.03.2015), 2011.

[56] *Advancing Moore's Law in 2014*, Intel Corporation, 2014.

[57] *Intel Core i7 Processor Family for LGA2011-v3 Socket*, Datasheet (Volume 1 of 2), Intel Coporation, August 2014.

[58] F. Irigoin, P. Jouvelot, and R. Triolet, *Semantical interprocedural parallelization: An overview of the PIPS project*, In Proceedings of the 1991 ACM International Conference on Supercomputing, Cologne Germany, June 1991.

[59] ISO/IEC 9899:1990, *Programming languages - C*, 1990.

[60] T. B. Jablin, *Automatic Parallelization for GPUs*, PhD thesis, Princeton University, 2013.

[61] X. Jia, X. Gu, J. Sempau, D. Choi, A. Majumdar, S. B. Jiang, *Development of a GPU-based Monte Carlo dose calculation code for coupled electron–photon transport*, Physics in medicine and biology, 55(11): 3077, 2010.

[62] L. Jóźwiak, *Life-inspired systems*, DSD'2004 - Euromicro Symposium on Digital System Design, 31 August - 3 September 2004, Rennes France, IEEE Computer Society Press, Los Alamitos, pages 36-43, 2004.

[63] L. Jóźwiak, N. Nedjah, *Modern architectures for embedded reconfigurable systems - a survey*, Journal of Circuits, Systems and Computers 18 (2), pages 209-254, 2009.

[64] L. Jóźwiak, N. Nedjah, M. Figueroa, *Modern development methods and tools for embedded reconfigurable systems: A survey*, INTEGRATION, the VLSI journal, vol. 43, pages 1-33, 2010.

[65] B. W. Kernighan, D. M. Ritchie, *The C programming language*, Vol. 2. Englewood Cliffs: prentice-Hall, 1988.

[66] B. Kienhuis, E. Rijpkema, E. F. Deprettere, *Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures*, 8th International Workshop on Hardware/Software Codesign (CODES'2000), San Diego, CA, USA, 2000.

[67] S. Kilts, *Advanced FPGA design: architecture, implementation, and optimization*, John Wiley & Sons, 2007.

[68] J. Knoop, O. Rüthing, B. Steffen, *Partial dead code elimination*, Vol. 29, No. 6, ACM, 1994.

[69] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[70] *LangPop - Programming Language Popularity*, `http://www.langpop.com` (abgerufen am 28.01.2015), 2013.

[71] C. Lapkowski, L. J. Hendren, *Extended SSA mumbering: Introducing SSA properties to languages with multi-level pointers*, Compiler Construction, Springer Berlin Heidelberg, 1998.

[72] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, M. S. Lam, *SUIF Explorer: an interactive and interprocedural parallelizer*, SIGPLAN Not. 34 (8), May 1999.

[73] K. C. Louden, *Compiler construction*, Cengage Learning, 1997.

[74] D. A. MacKenzie, *Knowing machines: Essays on technical change*, MIT Press, 1998.

[75] *Microsoft Visual Studio 2010*, Microsoft Corporation, `http://www.visualstudio.com` (abgerufen am 15.04.2015), 2010.

[76] *Automatische Parallelisierung*, Microsoft Visual Studio 2012 – Dokumentation, `http://www.microsoft.com` (abgerufen am 15.12.2014), 2014.

[77] E. F. Moore, *Machine models of self-reproduction*, Proceedings of Symposia in Applied Mathematics, volume 14, pages 17–33, The American Mathematical Society, 1962.

[78] G. E. Moore, *Cramming more components onto integrated circuits*, Eletronics, 38(8), pages 114-117, 1965.

[79] M. Mortensen, *High Level Parallel Programming Language Compiling to a Cellular Automata Processing Model*, M.Sc. Thesis, Aarhus Universitet, Datalogisk Institut, 2007.

[80] *NVIDIA GeForce GTX 580*, GPU Datasheet, NVIDIA Coporation, 2010.

[81] *NVIDIA Tesla K-Series*, Datasheet, NVIDIA Coporation, October 2013.

[82] OpenMP ARB, *OpenMP, A Proposed Industry Standard API for Shared Memory Programming*, 1997.

[83] D. A. Padua, M. J. Wolfe, *Advanced compiler optimizations for supercomputers*, Commun. ACM 29 (12), pages 1184-1201, December 1986.

[84] C. Paar, J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*, Springer Science & Business Media, 2009.

[85] D. A. Patterson, C. H. Sequin, *RISC I: A Reduced Instruction Set VLSI Computer*, In ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture, pages 443-457, IEEE Computer Society Press, 1981.

[86] D. A. Patterson, J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface (3rd ed.)*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[87] *PIPS: An interprocedural extensible source-to-source compiler infrastructure for code/application transformations and instrumentations*, In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11), IEEE Computer Society, Washington, DC, USA, 2011.

[88] L.-N. Pouchet, C. Bastoul, and U. Bondhugula, *PoCC: the Polyhedral Compiler Collection*, `http://pocc.sf.net` (abgerufen am 15.12.2014), 2010.

[89] *Qt Project*, `http://qt-project.org` (abgerufen am 02.02.2015), 2015.

[90] B. K. Rosen, M. N. Wegman, F. K. Zadeck, *Global value numbers and redundant computations*, In Proceedings of the 15th SIGPLAN-SIGACT symposium on Principles of programming languages, pages 12–27, 1988.

[91] R. M. Russell, *The CRAY-1 computer system*, Commun. ACM 21 (1), January 1978.

[92] P. W. Sathyanathan, *Interprocedural Dataflow Analysis - Alias Analysis*, Ph.D. thesis, Stanford University, Computer Systems Laboratory, June, 2001.

[93] S. Schmitt, *Integrierte Simulation und Emulation eingebetteter Hardware/Software-Systeme*, Dissertation, Göttingen : Cuvillier, 2005.

[94] B. Shneiderman, *Tree visualization with tree-maps: 2-d space-filling approach*, ACM Transactions on graphics (TOG), 11(1) ,pages 92-99, 1992.

[95] M. D. Smith , G. Holloway, *An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization*, Harvard University, 2002.

[96] S. Spacey, W. Luk, P. H. J. Kelly, D. Kuhn, *Improving communication latency with the write-only architecture*, Journal of Parallel and Distributed Computing, Volume 72, Issue 12, pages 1617-1627 December 2012.

[97] S. D. Stearns, *Digital Signal Analysis*, Hayden Book Company, Rochelle Park, New Jersey, 1975.

[98] J. Stein, *Computational problems associated with Racah algebra*, Journal of Computational Physics, vol. 1(3), pages 397-405, 1967.

[99] D. Talia, *Cellular automata + parallel computing = computational simulation*, Proc. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, Vol. 6, 1997.

[100] A. S. Tanenbaum, M. Van Steen, *Distributed systems*, Prentice-Hall, 2007.

[101] W. Thies, V. Chandrasekhar, and S. Amarasinghe, *A practical approach to exploiting coarse-grained pipeline parallelism in c programs*, Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2007.

[102] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, E. F. Deprettere, *A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs*, In CODES+ISSS '07: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, pages 9-14, ACM, New York, NY, USA, 2007.

[103] J. E. Thornton, *Design of a computer: the Control Data 6600*, Scott, Foresman, 1970.

[104] T. Toffoli, N. Margolus, *Cellular automata machines: a new environment for modeling*, MIT press, 1987.

[105] K. R. Traub, *Compilation as partitioning: a new approach to compiling non-strict functional languages*, In Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89), 1989.

[106] T. Ungerer, *Datenflußrechner*, Teubner-Verlag, 1993.

[107] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, *Polyhedral parallel code generation for CUDA*, ACM Trans. Archit. Code Optim. 9 (4), Article 54, January 2013.

[108] J. von Neumann, *First draft of a report on the EDVAC*, Technical report, University of Pennsylvania, 1945.

[109] J. von Neumann, *Theory of Self-Reproducing Automata*, Univ. of Illinois Press, 1966.

[110] K. Walus, T. J. Dysart, G. A. Jullien, R. A. Budiman, *QCADesigner: a rapid design and Simulation tool for quantum-dot cellular automata*, Nanotechnology, IEEE Transactions on, vol. 3, no. 1, pages 26-31, 2004.

[111] S. Wasson, *Nvidia's GeForce 8800 graphics processor*, Technical report, PC Hardware Explored, 2006.

[112] M. Weinhardt, W. Luk, *Pipeline Vectorization*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2001.

[113] M. E. Wolf, M. S. Lam, *A loop transformation theory and an algorithm to maximize parallelism*, IEEE Transactions on Parallel and Distributed Systems, 2(4), pages 452-471, 1991.

[114] A. Wolfe, *Intel Clears Up Post-Tejas Confusion*, VARBusiness Magazine, May 17 (2004), `http://www.crn.com/news/channel-programs/18842588/intel-clears-up-post-tejas-confusion.htm` (abgerufen am 12.11.2014), 2004.

[115] *Xilinx ISE Design Suite*, `http://www.xilinx.com/products/design-tools/ise-design-suite.html` (abgerufen am 25.03.2015), 2015.

[116] *Xilinx University Program XUPV5-LX110T Development System*, `http://www.xilinx.com/univ/xupv5-lx110t.htm` (abgerufen am 02.04.2015), 2015.

[117] K. Zuse, *Patentanmeldung Z-2391*, German Patent Office, Berlin, 1941.

[118] K. Zuse, *Rechnender Raum*, Vieweg Braunschweig, 1969.

[119] K. Zuse, *Der Computer - Mein Lebenswerk*, Springer-Verlag, Berlin, 1970.

Abbildungsverzeichnis

Tabellenverzeichnis